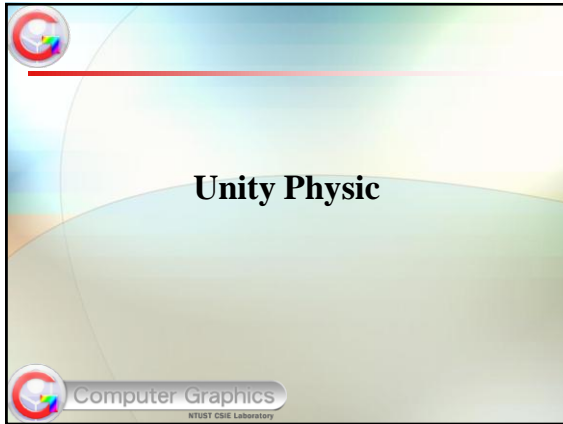


教材、教案、程式專案簡錄

手機遊戲設計教材節錄



物理元件使用與設定

- Unity內建的物理引擎可處理物體碰撞、布料模擬
- 大量物理元件同時使用容易拖慢效能
- 套用於元件時，可分作幾個項目
 - Collider(碰撞體)
 - 物體可進行碰撞邊界的大小、位置、形狀、碰撞材質設定
 - 常用的有方形(Box)、球形(Sphere)、膠囊型(Capsule)及地面(Terrain)碰撞體
 - Rigidbody(鋼體)
 - 任何物件都需要加入Rigidbody後才能變成動態物件，可以設定質量等基本係數
 - Physic Material(物理材質)
 - 摩擦係數、角摩擦係數、重力

Computer Graphics
NTUST CSIE Laboratory

找到場景中的物件

- 得到物件
 - `GameObject obj = GameObject.Find("ObjectName");`
 - `Camera cml = GameObject.Find("Main Camera").GetComponent<Camera>();`
- 得到物件上的component如Rigidbody
 - `Rigidbody rd = GameObject.Find("Cube2").GetComponent<Rigidbody>();`

Computer Graphics
NTUST CSIE Laboratory

物理引擎-移動一個物體的方法

- Rigidbody
 - `rigidbody.velocity = new Vector3(0,0,1.0f);`
 - `rigidbody.AddForce(new Vector3(0,0,1.0f));`
- Transform
 - `transform.Translate(Vector3.forward * Time.deltaTime, Space.World);`
 - `transform.position += Vector3.forward * Time.deltaTime;`

和上一次更新的時間差

Computer Graphics
NTUST CSIE Laboratory

物體旋轉及縮放

- 旋轉 (直接改變旋轉狀態)
 - `go.transform.rotation = Quaternion.Euler(0, r, 0);`
 - `go.transform.localRotation = Quaternion.Euler(0, r, 0);`
- 旋轉(根據目前狀態旋轉)
 - `go.transform.Rotate(new Vector3(0, r, 0));`
- 縮放
 - `go.transform.localScale = new Vector3(1,2,3);`

Computer Graphics
NTUST CSIE Laboratory

練習

- 建立一個Plane
- 放置4個cube，加上rigidbody，並使用不同的方式移動這些方塊

Computer Graphics
NTUST CSIE Laboratory

物理引擎-Rigidbody

- Gravity : 受不受重力影響(世界重力的設定在 Project Setting->Physics中的Gravity)
- Force : 外力
- Drag : 阻力
- Friction : 摩擦力 (由Collider的Material設定, 但通常需要enable Gravity, 才會有效果)
- Velocity : 速度 (由程式設定)
- Non-Kinematic vs Kinematic
 - Non-Kinematic : AddForce() and AddTorque()
 - Kinematic Rigidbodies : 不受外力、重力、碰撞 (但可以收到isTrigger事件) 影響。改變位置的方式需要直接更改position, 當自己移動, 碰撞到其他物體時, 仍可以造成對方的影響

Computer Graphics
NTUST CSIE Laboratory

練習

- 以剛剛的四個cube為基礎, 加入一道cube牆(在移動物體會經過的地方)
- 嘗試修改Cube的gravity, isKinematic (如用 translate/transform移動的cube加上isKinematic) 觀察物體移動的效果, 撞到? 穿過去?

Computer Graphics
NTUST CSIE Laboratory

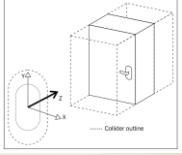
物理引擎-Collider

- Static collider : 只有collider, 則靜止不動, 讓別人撞, 但自己不會移動
- Rigidbody Collider: 同時有rigidbody and collider, 發生碰撞時, 會受物理引擎 (重力, 外力) 影響, 而適當移動
- Kinematic Rigidbody Collider : 包含rigidbody 跟 collider, rigidbody中enable isKinematic. 碰撞效果跟Static collider類似, **如果希望收到碰撞事件**, 又希望被碰撞時自己不受做用力影響, 則設定成 isKinematic
- unity3d建議: Colliders that move should always be Kinematic Rigidbodies.

Computer Graphics
NTUST CSIE Laboratory

物理引擎-Collision

- 碰撞的發生
 - 兩個物體如果都有碰撞體(Collider) 則在接觸的時候, 會發生碰撞事件, 視設定來決定行為
- 如有rigidbody (no kinematic)則會撞開, 並收到相關訊息事件, 如下面三個
 - void OnCollisionEnter(Collision collision)
 - void OnCollisionStay(Collision collision)
 - void OnCollisionExit(Collision collision)
- 摩擦力: 碰撞的設定包括一個physic Material, 可以決定反彈特性跟摩擦力.. 等等, 需要搭配rigidbody的gravity發生
- 要收到碰撞事件, 自己本身要有rigidbody



Computer Graphics
NTUST CSIE Laboratory

使用Layer

- 每個GameObject都有所屬的Layer, 預設為Default
- 設定方法在Inspector視窗中的Layer
- 加入新Layer的方法為Add Layer...之後會出現 TagManager
- Layer以數字存在




Layer Pullet的 Layer Mask ID 為8

Computer Graphics
NTUST CSIE Laboratory

設定互相碰撞的Layer

- 碰撞的設定方法為按下Edit->Project Setting->Physics, 在Inspector視窗內會出現如下圖
- 在Layer Collision Matrix中可自由設定layer之間的碰撞關係(是否會碰撞)



Pipe對上Pullet的欄位沒打勾代表這兩個Layer的物件互不相碰撞

Computer Graphics
NTUST CSIE Laboratory

物理引擎- Collision

- **重要屬性 Is Trigger :**
 - 可忽略物理引擎效果的碰撞效果，如反作用力、阻力
 - 要引發Trigger事件，要收訊息的要有rigidbody
 - 使用時間：在靠近門的時候，自動開門，走到特定的地方，顯示資訊
 - Enable isTrigger則收到的訊息不再是 OnCollisionEnter，而是


```
void OnTriggerEnter(Collider other)
void OnTriggerStay(Collider other)
void OnTriggerExit(Collider other)
```

Computer Graphics
NTUST CSIE Laboratory

物理引擎- Collision

- **Collider 的形狀類型**
 - Box Collider - 方塊碰撞體
 - Sphere Collider - 球狀碰撞體
 - Capsule Collider - 膠囊狀碰撞體
 - Mesh Collider - 使用model本身作為碰撞體，Mesh Collider 不能互相碰撞
 - Wheel Collider - 圓柱狀碰撞體

Computer Graphics
NTUST CSIE Laboratory

練習

- 以剛剛的練習為基礎，設定三道cube牆(在移動物體會經過的地方)
 - 第一道：Collider (isTrigger)
 - 第二道：Collider
 - 第三道：Collider + Rigidbody
- 嘗試修改移動cube的gravity，isKinematic，觀察物體移動的效果，撞到?穿過去?
- 修改四個script，都接收OnCollisionEnter跟 OnTriggerEnter事件，在裡面印出碰撞的物件的name
- isTrigger跟isKinematic都會忽略物理引擎效果，所以都不會收到OnCollisionEnter訊息

Computer Graphics
NTUST CSIE Laboratory

Question

- 坦克砲彈
 - Rigidbody (Gravity, AddForce, Velocity, IsKinematic),
 - Collider(Friction, isTrigger)
- 發射飛彈
 - Rigidbody (Gravity, AddForce, Velocity, IsKinematic),
 - Collider(Friction, isTrigger)
- 不會動的城牆
 - Rigidbody (Gravity, AddForce, Velocity, IsKinematic),
 - Collider(Friction, isTrigger)
- 會被主角碰到移動的垃圾桶
 - Rigidbody (Gravity, AddForce, Velocity, IsKinematic),
 - Collider(Friction, isTrigger)

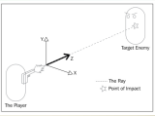
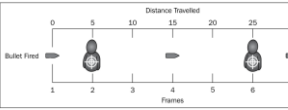
Computer Graphics
NTUST CSIE Laboratory

物理引擎-Raycast

觀念：像子彈的物體移動速度太快，超過物理引擎偵測碰撞的頻率 (frame rate) 所以可能誤判，所以使用預測方式的碰撞偵測

產生一個模擬的射線，沿著設定好的參數 (起始點，方位、長度) 射出，判斷是否碰撞到物體，如果有則傳回碰撞物體的資訊 (RaycastHit)

類型一：從物體的角度來偵測是否碰到物體，並預測是否會發生碰撞

Computer Graphics
NTUST CSIE Laboratory

Raycast最近的物體

```
RaycastHit hitInfo = new RaycastHit();
Vector3 dir = new Vector3(-1,0,0);
if(Physics.Raycast (this.transform.position, dir, out hitInfo, 1))
{
      
    作為起點

    if (hitInfo.collider.gameObject.name == "CubeA")
    {
        print("shoot");
    }
}
```

Computer Graphics
NTUST CSIE Laboratory

Raycast 所有物體

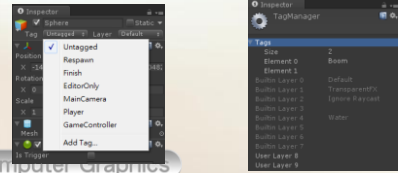
```
Vector3 dir = new Vector3(-1,0,0);
RaycastHit[] hitInfos =
Physics.RaycastAll(this.transform.position, dir);
foreach (RaycastHit hitInfo in hitInfos)
{
    print(hitInfo.collider.gameObject.name);
}
```

Computer Graphics

NTUST CSIE Laboratory

為物件標上Tag

- Tag旗標是用來識別和分類物件的方法之一
- 在Raycast時能用於識別物件
- 設定Tag的方法在Inspector視窗中的Tag中點擊Add Tag...之後會出現TagManager，拉下Tags之後就能加入新的Tag



Computer Graphics

NTUST CSIE Laboratory

針對某類(Tag)做Raycast 搜尋

```
Vector3 dir = new Vector3(-1,0,0);
RaycastHit[] hitInfos =
Physics.RaycastAll(this.transform.position, dir);
foreach (RaycastHit hitInfo in hitInfos)
{
    if (hitInfo.collider.gameObject.tag != "Boom")
        print(hitInfo.collider.gameObject.name);
}
```

Computer Graphics

NTUST CSIE Laboratory

物理引擎-Raycast

類型二：從Camera的角度來偵測是否碰到物體，若有，則可取到由滑鼠或手指點到的GameObject

範例

```
Vector3 pos = Input.mousePosition;
Ray mouseRay = Camera.main.ScreenPointToRay(pos);
if (Input.GetMouseButton(0))
{
    if (Physics.Raycast(mouseRay) )
    {
        ...
    }
}
```

Computer Graphics

NTUST CSIE Laboratory

練習

- 使用RayCast的技巧，在遊戲畫面中，按下滑鼠右鍵，點選任一個遊戲物件，當滑鼠點擊到方塊就讓方塊往上跳

Computer Graphics

NTUST CSIE Laboratory

物理元件使用與設定

- 新增剛體效果於基本物件上
 - 選擇一個基本物件之後點選上方選單的 [Component]→[Physics]→[Rigidbody]
 - 點選 播放後可看到物體掉至地面上後停止



Computer Graphics

NTUST CSIE Laboratory

物理元件使用與設定

- 可設定物體碰撞時的特性
- 選擇選單的[Assets]→[Import Package]→[Physic Materials]
- 之後按Import，將預設碰撞材質包全部匯入




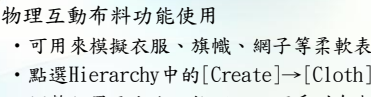
物理元件使用與設定

- 點選物件，點選[Component]項目中[Material]旁的圓圈，可選擇剛匯入的預設碰撞材質
- 套用之後點選  即可看到物件產生不同的碰撞效果



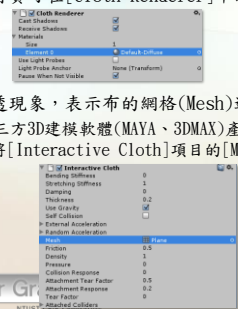
物理元件使用與設定

- 物理互動布料功能使用
 - 可用來模擬衣服、旗幟、網子等柔軟表面
 - 點選Hierarchy中的[Create]→[Cloth]
 - 調整位置及大小，按下  可看到布與之前產生的物件發生互動



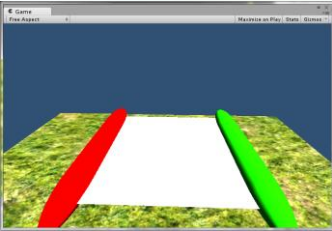
物理元件使用與設定

- 布的表面材質可在[Cloth Renderer]中的[Materials]項目設定
 - 若發生穿透現象，表示布的網格(Mesh)過於稀疏
 - 可由第三方3D建模軟體(MAYA、3DMAX)產生較緊密之網格平面，再將[Interactive Cloth]項目的[Mesh]設為匯入之平面模型



物理元件使用與設定

- 布料物裡綁定
 - 可將布料固定在多個其他物件上
 - 新增兩個長條物件於布的两端，作為固定的物體



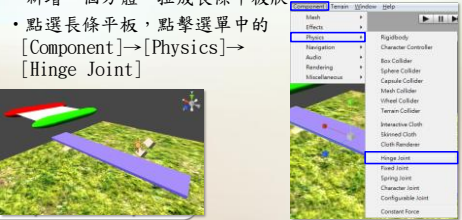
物理元件使用與設定

- 選擇布面，將[Interactive Cloth]項目中之[Attached Colliders]的Size設為2
- 分別將[Element 0]及[Element 1]中的[Collider]設為剛新增用來固定布的物件
- 之後按下  即可看到布掛在兩根物件上



物理元件使用與設定

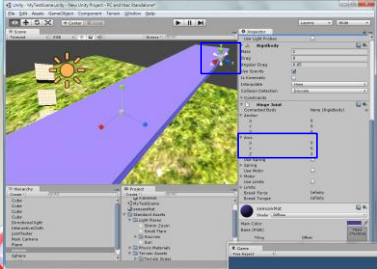
- 物理關節
 - 用以設定物件沿著某一軸心轉動
 - 以Hinge Joint為例：蹺蹺板
 - 新增一個方體，拉成長條平板狀
 - 點選長條平板，點選選單中的 [Component] → [Physics] → [Hinge Joint]



NTUST CSIE Laboratory

物理元件使用與設定

- 在Hinge Joint項目中設定Ancher至(0, 0, 0)物件中心點，Axis設為蹺蹺板旋轉的固定軸心

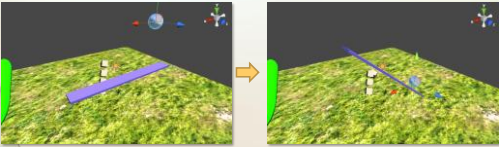


圖中旋轉軸心為Z軸，故Axis設為(0, 0, 1)

NTUST CSIE Laboratory

物理元件使用與設定

- 在蹺蹺板上方新增一個物件，並賦予Rigidbody特性
- 按下▶，物件下落後敲擊到蹺蹺板，即可看到蹺蹺板以剛剛設定好之軸心旋轉



Computer Graphics
NTUST CSIE Laboratory

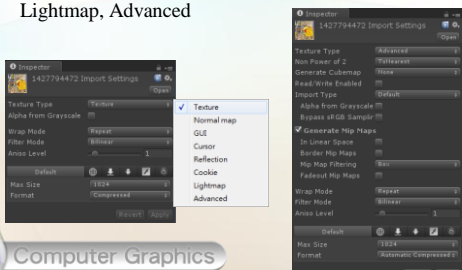
Import model & texture

- Model
 - Unity支援.FBX, .dae, .3DS, .dxf 和 .obj等Model格式
 - 加入Model方法：直接將model檔拉進Assets視窗內
- Texture
 - 幾乎所有圖檔格式都支援
 - 加入的方法跟Model相同

Computer Graphics
NTUST CSIE Laboratory

Import setting

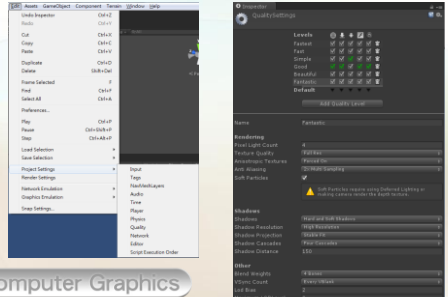
- Texture Type
 - Texture, Normal map, GUI, Cursor, Reflection, Cookie, Lightmap, Advanced



Computer Graphics
NTUST CSIE Laboratory

Project setting

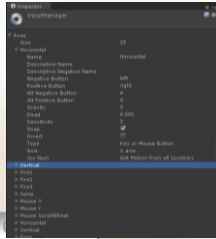
- Edit → Project Setting → Quality(品質)



Computer Graphics
NTUST CSIE Laboratory

Input setting

- Input Manager 中可以定義所有input axes和遊戲動作
- 打開的方式：Edit->Project Setting->Input



Computer Graphics

NTUST CSIE Laboratory

系統function

- Awake(){}：物件創建之初呼叫
- Start() {}：任何Update執行前，所有物件初始化完成之後執行
- Update() {}：每個frame更新時呼叫
- FixedUpdate() {}：固定的framerate下呼叫此函數
- OnGUI() {}：當GUI事件(按下滑鼠或鍵盤)發生和畫面更新時呼叫
- OnMouseDown(){}：當滑鼠按下時呼叫
- OnTriggerEnter(){}：當有Trigger物件**剛接觸**時呼叫
- 參考網站：
<http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.html>

Computer Graphics

NTUST CSIE Laboratory

常用系統物件

- Screen: 取得螢幕長和寬
ScreenW = Screen.width;
ScreenH = Screen.height;
- Application: 取得應用程式的run-time資料
Application.LoadLevel("Level1");
- Time: 在Unity中取得時間資訊的class.
Time.deltaTime
- GUI: GUI類別用於處理介面，包含按鍵和介面圖片
void OnGUI(){
if (GUI.Button(new Rect(100, 100, 200, 200),"Play")){
GUIHintHandle.iCountLoadingAnimation = 0;
}
}

Computer Graphics

NTUST CSIE Laboratory

常用系統物件及函式

- Input
 - Input.GetAxis("Vertical");
between -1 and 1 telling us which key is being pressed, pressing A will give us -1 and pressing D will give us 1.
 - if(Input.GetButtonDown("Fire1")){ }
- Instantiate();
 - GameObject abc = Instantiate(prefab);
- Camera
 - Camera objCamera = Camera.main.transform;
- Print(); debug 列印訊息
- Destroy(gameObject, t); t 秒後銷燬一個空間中的gameObject

Computer Graphics

NTUST CSIE Laboratory

Prefab

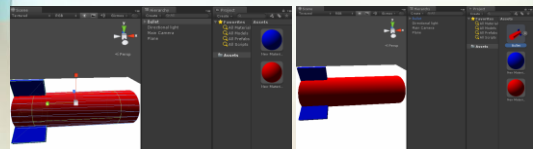
- Prefab
 - 可重複使用，且在遊戲執行中**動態增減**這些物件
 - 在場景中新加入一個Prefab物件時，相當於創建一個新的**實體(instance)**，所有Prefab的實體連結到原始的Prefab物件
 - 在程式執行中修改Prefab物件會改變所有使用相同Prefab創建的實體

Computer Graphics

NTUST CSIE Laboratory

Prefab

要創建一個Prefab物件只需要拖曳你希望重複使用的GameObject進入Project View中，拉入之後該GameObject的名字會變為藍色。



Computer Graphics

NTUST CSIE Laboratory

練習

- 練習: 讓Tank (Cube) , 發射一個子彈
 - 放一個plane , cube進入場景
 - 加入一個球並改變其顏色(材質)
 - 將剛加入的球建成prefab
 - 產生3個 GUI Buttons (right, left, front, back)
 - 新建一個script在Tank上並實現以下功能
 - 控制物體的移動 (左右移動)
 - 發射一個子彈 , 可以自我銷燬

Computer Graphics

NTUST CSIE Laboratory

sample code

- `GameObject Tank = GameObject.Find("Tank");`
- `GUI.Button(Rect(0,100,100,100) "Left");`
- `bullet= Instantiate (bulletPrefab, transform.position, Quaternion.identity);`
- `bullet.velocity = transform.TransformDirection(Vector3(0, 5, _ballSpeed));`
- `Physics.IgnoreCollision(transform.collider, bullet.transform.collider);`
//設定忽略碰撞
- `Destroy(gameObject, 2);`

Computer Graphics

NTUST CSIE Laboratory

GUI scripting guide

- GUI stands for Graphical User Interface.
- 使用現成元件
 - GUI Texture/GUIText
- 使用GUI API (需要放置在OnGUI () 中)
 - GUILDrawTexture
 - GUILLabel
 - GUILButton
 -
- GUISkin
- GUIStyle
- <http://unity3d.com/support/documentation/Components/GUI%20Scripting%20Guide.html>

Computer Graphics

NTUST CSIE Laboratory

GUI- GUI controls

- 顯示文字
`GUI.Label (new Rect (25, 25, 100, 30), "Label");`
- 畫2D圖片
`GUI.DrawTexture(new Rect(10,10,60,60), aTexture, ScaleMode.ScaleToFit, true, 10.0f);`
- 畫按鈕
`if (GUILButton (new Rect (10,10,150,100), "Button")) {`
 `print ("You clicked the button!");`
}

Computer Graphics

NTUST CSIE Laboratory

手機遊戲設計專案

Project 1：迷宮遊戲

◆ 目的：

經由本專題製作，熟悉 Unity3D 的基本操作及使用者介面，並嘗試以 C# 撰寫程式與場景內物件互動。

◆ 遊戲主軸：

開發一個 3D 的迷宮遊戲，使用者可透過方向鍵控制主角移動，並且美化場景及使用者介面。

◆ 基本項目：

1. 搭建完整的迷宮，並且使用正確的物理性質。
2. 使用正確的人物操作。
3. 簡單的使用者介面。
4. 畫面設計
 - A. 光源。
 - B. 粒子系統。
 - C. 地形。
 - D. 天空盒
 - E. 其他。
5. 聲音
 - A. 音效。。
 - B. 背景音樂。

◆ 進階功能：

1. 使用多重攝影機並且可以進行切換。
2. 使用小地圖。
3. 設置迷宮機關。
4. 遊戲性
 - A. 劇情。
 - B. AI。
 - C. 計時。
 - D. 場景切換。
 - E. 其他。
5. 其他創意。

Project 2：2D 多人遊戲

- ◆ 目的：
經由本專題製作，熟悉 Unity3D 的 2D、網路連線以及 AssetBundle 功能。
- ◆ 遊戲主軸：
開發一個 2D 的多人連線遊戲，玩家可以透過網路連線與多人同時進行遊戲，同時透過加載 AssetBundle 的方式來動態替換遊戲中素材。
- ◆ 基本項目：
 1. 3 種以上的 2D 物理。
 2. 2 種以上的 2D 動畫。
 3. 提供 UI 讓使用者選擇擔任 Host 還是 Client。
 4. 物件正確同步。
 5. 讀取 AssetBundle 檔替換遊戲中素材。
 6. 5 種以上的 Shader 特效。
- ◆ 進階功能：
 1. 遊戲性
 - A. 劇情。
 - B. AI。
 - C. 計時。
 - D. 場景切換。
 - E. 其他。
 2. 進階操作介面
 - A. 華麗的使用者介面。
 - B. 虛擬搖桿。
 - C. 圖片按鈕。
 - D. 其他。
 3. 其他創意。

Project 3：手機遊戲

- ◆ 目的：

經由本專題製作，熟悉 Unity3D 中於行動裝置上之特殊互動方式，其中包含 AR、多點觸控、陀螺儀等功能。
- ◆ 遊戲主軸：

開發一個可遊玩的手機遊戲，該遊戲需使用到題目要求之技術，並且提供存讀檔功能方便使用者儲存進度。
- ◆ 基本項目：
 1. 可遊玩的手機遊戲。
 2. 使用 AR。
 3. 運用到 Multi-touch。
 4. 運用到 G-Sensor 或 Gyroscope。
 5. 存讀檔功能。
- ◆ 進階功能：
 1. 遊戲性
 - A. 劇情。
 - B. AI。
 - C. 計時。
 - D. 場景切換。
 - E. 其他。
 2. 進階操作介面
 - A. 華麗的使用者介面。
 - B. 虛擬搖桿。
 - C. 圖片按鈕。
 - D. 其他。
 3. 其他創意

Project 4：第一人稱射擊遊戲

◆ 目的：

經由本專題製作，熟悉 Unreal Engine 的基本操作及使用者介面，並嘗試以藍圖功能撰寫程式與場景內物件互動。

◆ 遊戲主軸：

以 Unreal Engine 提供的第一人稱射擊遊戲範本開發出一款可以遊玩的第一人稱射擊遊戲，並且利用 Unreal Engine 的強大渲染引擎建置精緻的遊戲場景。

◆ 基本項目：

1. 玩家基本操作。
2. 改變子彈外觀。
3. 敵人會移動。
4. 敵人移動套用走路動畫
5. 敵人會攻擊。
6. 子彈碰撞後的事件處理(例：敵人扣血, etc...)。
7. 使用者介面。

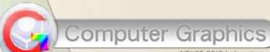
◆ 進階功能：

1. 敵人會動態生成。
2. 遊戲結束機制。
3. 額外音效。
4. 場景設計。
5. 其他創意。

圖學導論教材節錄

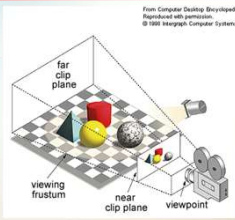

Last Note

- Visibility
 - Z-Buffer and transparency
 - A-buffer
 - Area subdivision
 - BSP Trees
 - Exact Cell-Portal



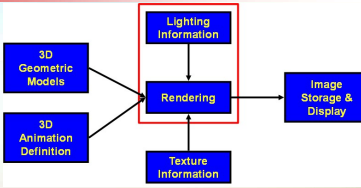

This Note

- Illumination Models
- Shading Models for Polygons
- Surface Detail
- The Rendering Pipeline
- Local Illumination and GL shading in Prof. Yao's Fundamental CG.
- Project 3
 - Check Point 1 Due at 11/21
 - Check Point 2 Due at 12/12
 - Demo Due at 1/16

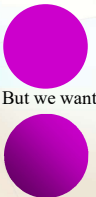
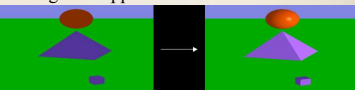

Where We Stand

- So far we know how to:
 - Transform between spaces
 - Draw polygons
 - Decide what's in front
- Next
 - Deciding a pixel's intensity and color






Why We Need Shading?

- Suppose we build a model of a sphere using many polygons and color it with **only one color**. We get something like
 - Human vision uses shading as a **cue** to form, position, and depth.
 - Total handling of light is very **expensive**.
- But we want
 - Shading models can give us a **good approximation** of what would "really" happen, much **less** expensively
 - Average and approximate

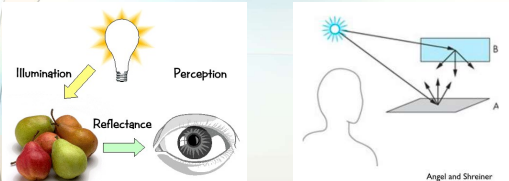




Shading

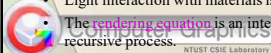
- Why does the image of a real sphere **look like**

 - **Light-material interactions** cause each point to have a different color or shade
 - Need to consider
 - Light sources
 - Material properties
 - Location of viewer
 - Surface orientation



Light Transport



- The most general approach is based on **physics** - using principles such as conservation of energy.
- A surface either **emits** light (e.g., light bulb) or reflects light for other illumination sources, or both.
- Light interaction with materials is **recursive**.
- The **rendering equation** is an integral equation describing the limit of this recursive process.



What Are the Patterns of Light in This Room?

- **Projector** as light source
- Light transmitted through **windows**
- Blackboard is **matte** surface
- **Edge** of screen is shiny surface
- **Shadows** underneath the desks

Computer Graphics 09/16/2010 © 2010 NTUST

Shading: Illumination

- Light Sources **emit light**
 - EM spectrum
 - Position and direction
- Surfaces **reflect light**
 - Reflectance
 - Geometry (position, orientation, micro-structure)
 - Absorption
 - Transmission
- Illumination determined by the **interactions between light sources and surfaces**

Computer Graphics NTUST CSIE Laboratory

Illumination (Shading) Models

- Interaction **between light sources and objects** in scene that results in **perception of intensity and color** at eye
- **Local vs. global** models
 - **Local**: perception of a particular primitive **only** depends on **light sources directly** affecting that one primitive
 - **Global**: also **take into account indirect effects** on light of other objects in the scene

Computer Graphics NTUST CSIE Laboratory

Local vs. Global Models

Local

- Geometry
- Material properties
- Shadows cast (global?)

Global

- Light reflected/refracted
- Indirect lighting

Computer Graphics NTUST CSIE Laboratory

Local vs. Global Models

Direct Lighting Only

Direct + Indirect Lighting

Computer Graphics NTUST CSIE Laboratory

Local vs. Global Models

Computer Graphics NTUST CSIE Laboratory

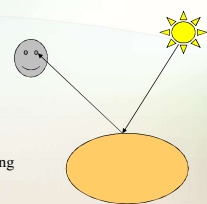
Local Shading Models

- Local shading models provide a way to determine the intensity and color of a point on a surface
 - The models are local because they **don't consider other objects**
 - We use them because they are **fast and simple** to compute
 - They do not require **knowledge of the entire scene**, only the current piece of surface. Why is this good for hardware?
- For the moment, assume:
 - We are applying these computations at a particular point on a surface
 - We have a **normal vector** for that point

Computer Graphics NTUST CSIE Laboratory 09/16/2010 © 2010 NTUST

Local Shading Models

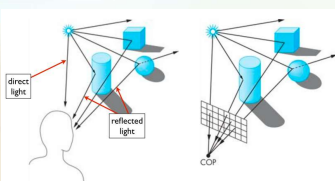
- The rendering equation can't be solved analytically
- Numerical methods aren't fast enough for real-time
- What they **capture**:
 - Direct illumination from light sources
 - Diffuse and Specular reflections (Phong reflection Model)
 - (Very) Approximate effects of global lighting
- What they **don't do**:
 - Shadows
 - Mirrors
 - Refraction



Computer Graphics NTUST CSIE Laboratory 09/16/2010 © 2010 NTUST

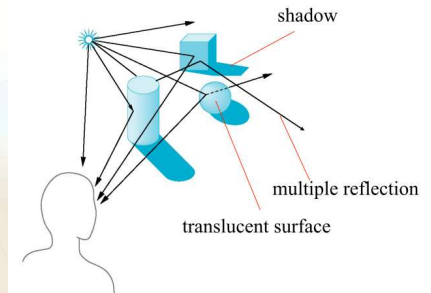
Local Shading Models

- Direct light is the color of the light source
- Reflected light is the color of the light reflected from the object surface.
- For rendering, **color of light source and reflected light** determines the colors of pixels in the frame buffer
- Only need to **consider the rays** that leave the source and reach the viewers eye



Computer Graphics NTUST CSIE Laboratory

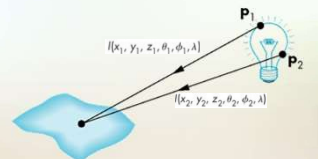
Global Effects



Computer Graphics NTUST CSIE Laboratory

Light Sources

- General light sources are **difficult to work** with because we must integrate **light coming from all points** on the source
- Illumination function: $I(x, \omega, \lambda)$

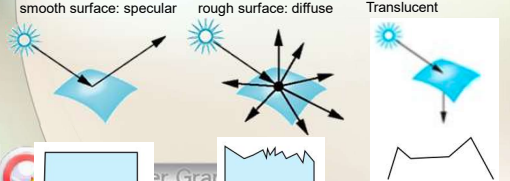


Detailed is given later

Computer Graphics NTUST CSIE Laboratory

Light-material Interactions

- At a surface, light is **absorbed, reflected, or transmitted**
- The smoother a surface, the more reflected light is **concentrated in the direction** a perfect mirror would reflected the light
- A very **rough** surface scatters light in **all directions**
- Translucent** allows some light to **pass through object** i.e. refraction and e.g., glass or water



Computer Graphics NTUST CSIE Laboratory

Surface Reflection

- When light hits an **opaque surface** some is **absorbed**,
- The rest is reflected (some can be transmitted too--but ignore that for now)
- The **reflected light** is what we see
- Reflection is **not simple** and varies with material
 - The surface's **micro structure** defines the details of reflection
 - Variations produce anything from **bright specular reflection** (mirrors) to dull matte finish (chalk)

Computer Graphics
NTUST CSIE Laboratory

Phong Reflection Model

- A **simple** model that can be computed rapidly
- Has **three components**
 - Diffuse
 - Specular
 - Ambient
- Uses **four vectors**
 - To source
 - To viewer
 - Normal
 - Perfect reflector

Ambient + Diffuse + Specular = Phong Reflection
Brad Smith, Wikimedia Commons

"Standard" Lighting Model

- Consists of three terms linearly combined:
 - Diffuse** component for the amount of incoming light from a point source reflected equally in all directions
 - Specular** component for the amount of light from a point source reflected in a mirror-like fashion
 - Ambient** term to approximate light arriving via other surfaces

Computer Graphics
NTUST CSIE Laboratory
09/18/2010 © 2010 NTUST

Ambient Shading

- Add constant color** to account for disregarded illumination and fill in black shadows; a cheap hack.

$$I = I_a k_a$$

- I_a : intensity of the ambient light
- k_a : ambient-reflection coefficient: 0 ~ 1

Computer Graphics
NTUST CSIE Laboratory

Diffuse Shading

Lambert's cosine law

direct: maximum light intensity
indirect: reduced light intensity

- The light is reduced by **cos of angle**
 - This is because same amount of light is spread **over larger area** when light comes in at an angle

Computer Graphics
NTUST CSIE Laboratory

Diffuse Shading

- Assume light **reflects equally** in all directions
 - Therefore surface looks same color from all views; "view independent"
- Known as **Lambertian and Matte**

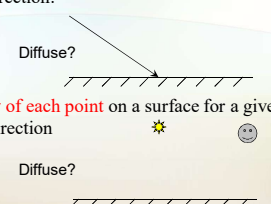
$$I = I_p k_d (\vec{N} \cdot \vec{V}_L)$$

- I_p : point light source's intensity
- k_d : diffuse-reflection coefficient: 0 ~ 1
- θ : angle: 0° ~ 90°
- v_L : direction to the light source
- n : surface normal
- Don't want to illuminate back side. Use $k_d I_i \max(L \cdot N, 0)$

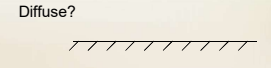
Computer Graphics
NTUST CSIE Laboratory

Illustrating Shading Models

- Show the **polar graph** of the amount of light leaving for a given incoming direction:



- Show the **intensity of each point** on a surface for a given light position or direction



Computer Graphics NTUST CSIE Laboratory 09/16/2010 © 2010 NTUST

Light-Source Attenuation

$$I = I_a k_a + f_{att} I_p k_d (\vec{N} \cdot \vec{L})$$

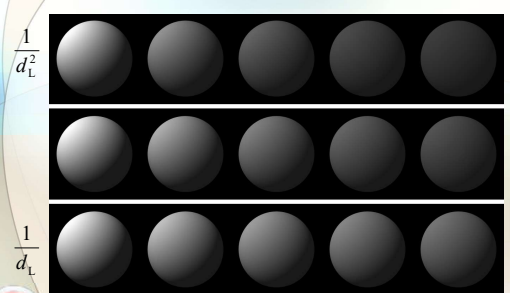
- f_{att} : **light-source attenuation factor**
 - if the light is a **point source**

$$f_{att} = \frac{1}{d_L^2}$$
 - where d_L is the distance the light travels from the point source to the surface

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right)$$

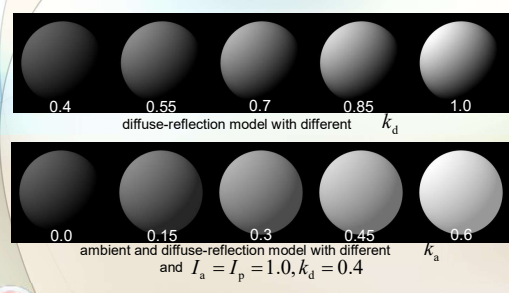
Computer Graphics NTUST CSIE Laboratory

Examples (Light Attenuation)



Computer Graphics NTUST CSIE Laboratory


Examples (k_d and k_a)



Computer Graphics NTUST CSIE Laboratory

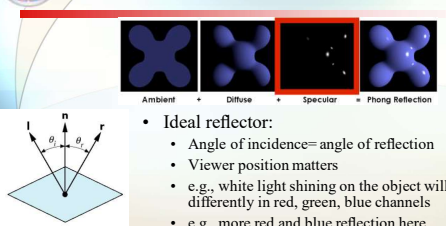
Specular Shading

- Some surfaces have highlights, mirror like reflection; view direction dependent; especially for smooth shiny surfaces



Computer Graphics NTUST CSIE Laboratory

Specular Reflection



- Ideal reflector:**
 - Angle of incidence = angle of reflection
 - Viewer position matters
 - e.g., white light shining on the object will be reflected differently in red, green, blue channels
 - e.g., more red and blue reflection here

Computer Graphics NTUST CSIE Laboratory

Specular Reflection (Phong Reflectance Model)

$k_s I_i (\mathbf{R} \cdot \mathbf{V})^p$

- Incoming light is reflected primarily in the mirror direction, \mathbf{R}
 - Perceived intensity depends on the relationship between the viewing direction, \mathbf{V} , and the mirror direction
 - Bright spot is called a *specularity*
- Intensity controlled by:
 - The specular reflectance coefficient, k_s
 - The *Phong Exponent*, p , controls the apparent size of the specularity
 - Higher n , smaller highlight

Computer Graphics
NTUST CSIE Laboratory

Specular Reflection

$I_s = k_s L_s \max(0, \cos \phi)^\alpha$

shininess coefficient

$\alpha = 5..10$ plastic
 $\alpha = 100..200$ metal

- Phong proposed this model
- Clamp to 0 -- avoid negative values
- The fuzzy highlight was too big without an exponent

Computer Graphics
NTUST CSIE Laboratory

Illustrating Shading Models

- Show the **polar graph** of the amount of light leaving for a given incoming direction:
- Show the **intensity of each point on a surface** for a given light position or direction

Computer Graphics
NTUST CSIE Laboratory

Specular Surfaces

- Most surfaces are **neither ideal diffusers nor perfectly specular** (ideal reflectors)
- Smooth surfaces show **specular highlights** due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection

Computer Graphics
NTUST CSIE Laboratory

The Phong Illumination Model

Computer Graphics
NTUST CSIE Laboratory

Examples (k_s and n)

k_s

0.1
0.25
0.5

$n = 3.0$ $n = 5.0$ $n = 10.0$ $n = 27.0$ $n = 200.0$

Computer Graphics
NTUST CSIE Laboratory

Examples (k_s and n)

p
 10: eggshell
 100: shiny
 1000: glossy
 10000: mirror-like

Computer Graphics
 NTUST CSIE Laboratory

Calculating the Reflection Vector

- Fall off gradually from the perfect reflection direction

$$\vec{R} = \vec{N} \cos \theta + \vec{S}$$

$$= \vec{N} \cos \theta + \vec{N} \cos \theta - \vec{L}$$

$$= 2\vec{N} \cos \theta - \vec{L}$$

$$= 2\vec{N}(\vec{N} \cdot \vec{L}) - \vec{L}$$

Computer Graphics
 NTUST CSIE Laboratory

The Halfway Vector (Blinn-Phong)

- Rather than computing reflection directly; just **compare to normal bisection property**.

$$\vec{H} = \frac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|}$$

$$\Rightarrow \cos \alpha \approx \vec{N} \cdot \vec{H}$$

Blinn-Phong Phong Blinn-Phong (Lower Exponent)

Computer Graphics
 NTUST CSIE Laboratory

Specular Reflection Improvement (Blinn-Phong)

$$\vec{H} = (\vec{L} + \vec{V}) / \|\vec{L} + \vec{V}\|$$

$$k_s I_i (\vec{H} \cdot \vec{N})^p$$

- Compute based on normal vector and "halfway" vector, **H**
- Always positive when the light and eye are above the tangent plane
- Not quite the same result as the other formulation (need 2H)

Computer Graphics 09/16/2010 © 2010 NTUST
 NTUST CSIE Laboratory

Putting It Together

$$I = k_a I_a + I_i (k_d (\vec{L} \cdot \vec{N}) + k_s (\vec{H} \cdot \vec{N})^p)$$

- Global ambient intensity, I_a :
 - Gross approximation to light bouncing around of all other surfaces
 - Modulated by ambient reflectance k_a
- Just sum all the terms
- If there are multiple lights, sum contributions from each light
- Several variations, and approximations ...

Computer Graphics
 NTUST CSIE Laboratory

The Phong Illumination Model

- $I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} \cos \theta + W(\theta) \cos^n \alpha]$
 - $W(\theta) = k_s$: specular-reflection coefficient: 0~1
- so, the Eq. can be rewritten as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}) + k_s (\vec{R} \cdot \vec{V})^n]$$
- consider the object's specular color

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}) + k_s O_{s\lambda} (\vec{R} \cdot \vec{V})^n]$$
 - $O_{s\lambda}$: specular color

Computer Graphics
 NTUST CSIE Laboratory

Example

The diagram illustrates the additive components of shading for a character. It shows four stages of shading: 1. Ground shading (a flat blue character), 2. Ambient (a slightly darker blue character), 3. Diffuse (a character with a gradient of blue), and 4. Specular (a character with a dark blue silhouette). The stages are separated by plus signs, indicating they are summed together.

Ground shading ambient diffuse specular

Computer Graphics
NTUST CSIE Laboratory

Approximations for Speed

- The viewer direction, V , and the light direction, L , depend on the surface position being considered, x
- Distant light approximation:**
 - Assume L is constant for all x
 - Good approximation if light is distant, such as sun
- Distant viewer approximation**
 - Assume V is constant for all x
 - Rarely good, but only affects specularities

Computer Graphics
NTUST CSIE Laboratory

Distant Light Approximation

- Distant light approximation:
 - Assume L is constant for all x
 - Good approximation if light is distant, such as sun
 - Generally called a *directional light source*
- What aspects of surface appearance are affected by this approximation?
 - Diffuse?
 - Specular?

Computer Graphics
NTUST CSIE Laboratory

Local Viewer Approximation

- Specularities require the viewing direction:
 - $V(x) = \|c-x\|$
 - Slightly expensive to compute
- Local viewer approximation uses a global V**
 - Independent of which point is being lit
 - Use the view plane normal vector
 - Error depends on the nature of the scene
- Is the diffuse component affected?

Computer Graphics
NTUST CSIE Laboratory

Light Sources

- Two aspects of light sources are important for a local shading model:
 - Where is the light coming from (the L vector)?
 - How much light is coming (the I values)?
- Various light source types give different answers to the above questions:
 - Point light source:** Light from a specific point
 - Directional:** Light from a specific direction
 - Spotlight:** Light from a specific point with intensity that depends on the direction
 - Area light:** Light from a continuum of points (later in the course)

luminance: $I = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}$

The diagram shows three types of light sources: 1. Point light: a small blue sphere emitting light rays from a single point. 2. Directional light: parallel light rays coming from a specific direction. 3. Area light: a blue rectangular surface emitting light rays from multiple points.

Computer Graphics
NTUST CSIE Laboratory

Simple Light Sources

- Point light**
 - Model with position and color
 - Distant source = infinite distance away (parallel) \Rightarrow **directional light**
- Ambient light**
 - Same amount of light everywhere in scene
 - Can model contribution of many sources and reflecting surfaces

Computer Graphics
NTUST CSIE Laboratory

Types of Light Sources

- Ambient: equal light in all directions
– a hack to model inter-reflections
- Directional: light rays oriented in same direction
– good for distance light sources (sunlight)
- Point: light rays diverge from a single point
– approximation to a light bulb (but harsher)

Computer Graphics
NTUST CSIE Laboratory

More Light Sources

- Spot light: point source with directional fall-off
– Restrict light from ideal point source
– Intensity is maximal along some direction D, falls off away from D
– Specified by color, point, direction, fall-off parameters
- Area light: Luminous 2D surface
– Radiates light from all points on its surface
– Generates soft shadows

Computer Graphics
NTUST CSIE Laboratory

Ambient Light Source

- Achieve a uniform light level
- No black shadows
- Ambient light intensity at each point in the scene

$$\mathbf{I}_a = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$$

I_a

Computer Graphics
NTUST CSIE Laboratory

Point Light Source

$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_r(\mathbf{p}_0) \\ I_g(\mathbf{p}_0) \\ I_b(\mathbf{p}_0) \end{bmatrix}$$

$$i(\mathbf{p}, \mathbf{p}_0) = \frac{1}{|\mathbf{p} - \mathbf{p}_0|^2} \mathbf{I}(\mathbf{p}_0)$$

- Illumination intensity at p:
- Use scalar $I(\vec{V}_{p_0})$ to denote any of three components.
- Points sources alone aren't too realistic looking -- tend to be high contrast => Add ambient light to mitigate high contrast
- Most real-world scenes have large light sources

Computer Graphics
NTUST CSIE Laboratory

Point Light Source

- Point light sources alone aren't too realistic
- Drop off intensity more slowly

$$i(\mathbf{p}, \mathbf{p}_0) = \frac{1}{d^2} \mathbf{I}(\mathbf{p}_0)$$

$$i(\mathbf{p}, \mathbf{p}_0) = \frac{1}{a + bd + cd^2} \mathbf{I}(\mathbf{p}_0)$$

- In practice, we also replace the $\frac{1}{d^2}$ term by something that falls off more slowly

Computer Graphics
NTUST CSIE Laboratory

Direct Light Source

- Most shading calculations require direction from the surface point to the light source position if the light source is very far, the direction vectors don't change e.g., sun
- Characterized by direction rather than position

Computer Graphics
NTUST CSIE Laboratory

OpenGL Point and Directional Sources

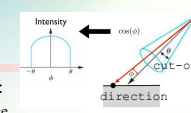
- Point light: $L(x) = \frac{P_{light} - x}{\|P_{light} - x\|}$
 - The **L vector** depends on where the surface point is located
 - Must be normalized - slightly expensive
 - To specify an OpenGL light at 1,1,1:


```
Gfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```
- Directional light: $L(x) = L_{light}$
 - The **L vector does not change** over points in the world
 - OpenGL light traveling in direction 1,1,1 (**L** is in opposite direction):


```
Gfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Computer Graphics
NTUST CSIE Laboratory

Spotlights



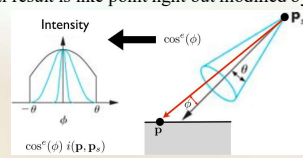
- Point source, but intensity depends on **L**:
 - Requires a **position**: the location of the source
`glLightfv(GL_LIGHT0, GL_POSITION, light_posn);`
 - Requires a **direction**: the center axis of the light
`glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light_dir);`
 - Requires a **cut-off**: how broad the beam is
`glLightfv(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);`
 - Requires an exponent: how the light tapers off at the edges of the cone
 - Intensity scaled by $(L \cdot D)^n$

```
glLightfv(GL_LIGHT0, GL_SPOT_EXPONENT, 1.0);
```

Computer Graphics
NTUST CSIE Laboratory

Spotlight Attenuation

- Add an exponent for greater control
 - Spotlight is brightest along \vec{l}_s
 - Vector \vec{v} with angle f from \mathbf{p} to point on surface
 - Intensity determined by $\cos(f)$
 - Corresponds to projection of \vec{v} onto \vec{l}_s
 - Spotlight exponent e determines rate of dropoff
- Final result is like point light but modified by this cone



Computer Graphics
NTUST CSIE Laboratory

Color

$$I_r = k_{a,r} I_{a,r} + I_{i,r} (k_{d,r} (\mathbf{L} \cdot \mathbf{N}) + k_{s,r} (\mathbf{H} \cdot \mathbf{N})^n)$$

- Do everything for three colors, **r, g and b**
- Note that some terms (the expensive ones) are constant
- Using only three colors is an **approximation**, but few graphics practitioners realize it
 - k terms depend on **wavelength**, should compute for continuous spectrum
 - Aliasing in color space
 - Better results use 9 color samples

Computer Graphics 09/16/2010 © 2010 NTUST
NTUST CSIE Laboratory

Colored Lights and Surfaces

- If an object's **diffuse color** is $O_d = (O_{dR}, O_{dG}, O_{dB})$ then $I = (I_R, I_G, I_B)$ where for the red component

$$I_R = I_{aR} k_a O_{dR} + f_{att} I_{pR} k_d O_{dR} (\vec{N} \cdot \vec{L})$$
 however, it should be

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\vec{N} \cdot \vec{L})$$
 where λ is the **wavelength**

Computer Graphics
NTUST CSIE Laboratory

Describing Surfaces

- The **various parameters** in the lighting equation describe the appearance of a surface
- $(k_{d,r}, k_{d,g}, k_{d,b})$: The **diffuse color**, which most closely maps to what you would consider the "color" of a surface
 - Also called *diffuse reflectance coefficients*
- $(k_{s,r}, k_{s,g}, k_{s,b})$: The **specular color**, which controls the color of specularities
 - The same as the diffuse color for metals, white for plastics
 - Some systems do not let you specify this color separately
- $(k_{a,r}, k_{a,g}, k_{a,b})$: The **ambient color**, which controls how the surface looks when not directly lit
 - Normally the same as the diffuse color

Computer Graphics 10/27/09 © NTUST
NTUST CSIE Laboratory

OpenGL Model

- Allows **emission, E** : Light being **emitted** by surface
- Allows **separate light** intensity for **diffuse and specular**
- Ambient light can be associated **with light sources**
- Allows **spotlights** that have intensity that depends on outgoing light direction
- Allows **attenuation** of light intensity with **distance**
- Can specify coefficients in multiple ways
- Too many variables** and commands to present in class
- The OpenGL programming guide goes through it all (the red book)

Computer Graphics 10/27/09 © NTUST

OpenGL Commands (1)

- `glMaterial{if}(face, parameter, value)`
 - Changes one of the coefficients **for the front or back side of a face** (or both sides)
- `glLight{if}(light, property, value)`
 - Changes one of the properties of a light (intensities, positions, directions, etc)
 - There are **8 lights**: `GL_LIGHT0`, `GL_LIGHT1`, ...
- `glLightModel{if}(property, value)`
 - Changes one of the global light model properties (global ambient light, for instance)
- `glEnable(GL_LIGHT0)` enables `GL_LIGHT0`
 - You **must enable lights** before they contribute to the image
 - You can enable and disable lights at any time

Computer Graphics 10/27/09 © NTUST

OpenGL Commands (2)

- `glColorMaterial(face, mode)`
 - Causes a material property**, such as diffuse color, to track the current `glColor()`
 - Speeds things up, and makes coding easier
- `glEnable(GL_LIGHTING)` **turns on lighting**
 - You **must enable lighting explicitly** – it is off by default
- Don't** use specular intensity if you don't have to
 - It's **expensive** - turn it off by giving 0,0,0 as specular color of the lights
- Don't forget normals**
 - If you use scaling transformations, must **enable `GL_NORMALIZE`** to keep normal vectors of unit length
- Many other things to control appearance

Computer Graphics 10/27/09 © NTUST

Multiple Light Sources

- If there are m light sources, then

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \leq i \leq m} f_{att_i} I_{p\lambda} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}_i) + k_s O_{s\lambda} \cos^n \alpha_i]$$

$$\approx I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \leq i \leq m} f_{att_i} I_{p\lambda} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}_i) + k_s O_{s\lambda} (\vec{R}_i \cdot \vec{V})^n]$$

$$\approx I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \leq i \leq m} f_{att_i} I_{p\lambda} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}_i) + k_s O_{s\lambda} (\vec{N} \cdot \vec{H}_i)^n]$$

Computer Graphics 10/27/09 © NTUST

Shading so Far

- So far, we have discussed **illuminating** a single point

$$I = k_a I_a + I_l (k_d (\mathbf{L} \cdot \mathbf{N}) + k_s (\mathbf{H} \cdot \mathbf{N})^p)$$
- We have assumed that we know:
 - The point
 - The surface normal
 - The viewer location (or direction)
 - The light location (or direction)
- But commonly, **normal vectors** are only given at the vertices
- It is also expensive to compute lighting for every point

Computer Graphics 10/27/09 © NTUST

Shading Polygonal Geometry

Computer Graphics 10/27/09 © NTUST

Shading Interpolation

- Take information **specified or computed at the vertices**, and somehow propagate it across the polygon (triangle)
- Several options:
 - Flat shading
 - Gouraud interpolation
 - Phong interpolation

Computer Graphics 10/27/09 © NTUST

Computing Lighting at Each Pixel

- Most accurate approach:** Compute component illumination at each pixel with individual positions, light directions, and viewing directions
- But this could be **expensive...**

Computer Graphics

Shading Models for Polygons

- Flat Shading**
 - Faceted Shading
 - Constant Shading
- Gouraud Shading**
 - Intensity Interpolation Shading
 - Color Interpolation Shading
- Phong Shading**
 - Normal-Vector Interpolation Shading

Computer Graphics

Flat Shading

- Compute shading at a **representative point** and apply to whole polygon i.e. apply illumination model once for each polygon.
 - OpenGL uses one of the vertices
- Assumptions
 - The light source is at infinity i.e. $\vec{l} \cdot \vec{n} = \text{constant}$
 - The viewer is at infinity i.e. $\vec{v} \cdot \vec{n} = \text{constant}$
 - The polygon **represents the actual surface** being modeled and is not an approximation to a curved surface
 - If light source or viewer is not at infinity, *need heuristic for picking color* - e.g., first vertex, or polygon center
 - does not produce variations in gradation

Computer Graphics

Flat Shading

- Advantages:
 - Fast** - one shading computation per polygon
- Disadvantages:
 - Inaccurate**
 - What are the artifacts?

Computer Graphics

Flat Shading and Perception

- Lateral inhibition: exaggerates perceived intensity
- Mach bands: perceived "stripes" along edges

Figure 6.28 Step chart.

Figure 6.29 Perceived and actual intensities at an edge.

Computer Graphics

Flat Shading

- Compute constant shading function, over each polygon
- Same normal and light vector across whole polygon
- Constant shading for polygon

$$I_p = I$$

Computer Graphics
NTUST CSIE Laboratory

Shading Polygons

- Polygons often approximate curve surfaces but are inherently flat
- Consider polygonal 'sphere'
- Want to smooth the rough face of each surface facet
- How do we fix this?

Computer Graphics
NTUST CSIE Laboratory

Smooth Shading

- We can simply find a new normal at each vertex for a sphere
- Easy for sphere model
 - If centered at origin $n = p$
- Results in smoother shading
- Note silhouette edge

Computer Graphics
NTUST CSIE Laboratory

Gouraud Shading

- Shade each *vertex* with it's own location and normal
- *Linearly interpolate* the color across the face
- Advantages:
 - **Fast**: incremental calculations when rasterizing
 - **Much smoother** - use same normal every time a vertex is used for a face
- Disadvantages:
 - What are the artifacts?
 - Is it accurate?

Computer Graphics
NTUST CSIE Laboratory

Intensity Interpolation (Gouraud)

$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3}$$

$$I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}$$

Computer Graphics
NTUST CSIE Laboratory

Phong Interpolation

- Interpolate **normals** across faces
- Shade each pixel individually
- Advantages:
 - High quality, narrow **specularities**
- Disadvantages:
 - Expensive
 - Still an approximation for most surfaces
- Not to be confused with **Phong's specularity** model

Computer Graphics
NTUST CSIE Laboratory

Normal Interpolation (Phong)

Computer Graphics
NTUST CSIE Laboratory

What is Normal?

Computer Graphics
NTUST CSIE Laboratory

Recall: Normal for Triangle

- Plane
 - $N \propto (P - P_2)$
 - $N = P_1P_2 \times P_1P_3$
 - $= (P_3 - P_1) \times (P_2 - P_1)$
- Normalize
 - $N \leftarrow N / |N|$
- Note that
 - right-hand rule determines outward face

Computer Graphics
NTUST CSIE Laboratory

Using Average Normals

Computer Graphics
NTUST CSIE Laboratory

Using Average Normals

Computer Graphics
NTUST CSIE Laboratory

Using Average Normals

$$\bar{N} = \frac{1}{2}(N_1 + N_2)$$

Computer Graphics
NTUST CSIE Laboratory

Using Average Normals

$$N_v = \frac{(N_1 + N_2 + N_3 + N_4)}{\|N_1 + N_2 + N_3 + N_4\|}$$

More generally,

$$N_v = \frac{\sum_{i=1}^n N_i}{\left\| \sum_{i=1}^n N_i \right\|}$$

It can also be area-weighted.

Computer Graphics
NTUST CSIE Laboratory

Definitions of Triangle Meshes

- $\{f_1\} : \{v_1, v_2, v_3\}$ connectivity
- $\{f_2\} : \{v_3, v_2, v_4\}$ connectivity
- ...
- $\{v_i\} : (x, y, z)$ geometry
- $\{v_2\} : (x, y, z)$ geometry
- ...
- $\{f_1\} : \text{"skin material"}$ face attributes
- $\{f_2\} : \text{"brown hair"}$ face attributes
- ...
- $\{v_2, f_1\} : (n_x, n_y, n_z) (u, v)$ corner attributes
- $\{v_2, f_2\} : (n_x, n_y, n_z) (u, v)$ corner attributes
- ...

Copyright©1998, Microsoft

Computer Graphics
NTUST CSIE Laboratory

Normal Interpolation (Phong)

$$N_a = N_1 \frac{y_s - y_2}{y_1 - y_2} + N_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$N_b = N_1 \frac{y_s - y_3}{y_1 - y_3} + N_3 \frac{y_1 - y_s}{y_1 - y_3}$$

$$\tilde{N}_p = \frac{N_a}{\|N_a\|} \left[\frac{x_b - x_p}{x_b - x_a} \right] + \frac{N_b}{\|N_b\|} \left[\frac{x_p - x_a}{x_b - x_a} \right]$$

$$N_p = \frac{\tilde{N}_p}{\|\tilde{N}_p\|} \quad \text{Normalizing makes this a unit vector}$$

Computer Graphics
NTUST CSIE Laboratory

Examples (1/2)

Gouraud

Phong

Computer Graphics
NTUST CSIE Laboratory

Examples (2/2)

Gouraud shading

Phong shading

Computer Graphics
NTUST CSIE Laboratory

Comparison (1/3)

Wire-frame

Flat Shading

Gouraud Shading

Phong Shading

Computer Graphics
NTUST CSIE Laboratory

Comparison (2/3)

- Phong interpolation looks smoother -- can see edges on the Gouraud model
- But Phong is a lot more work
 - both Phong and Gouraud require vertex normals
 - both Phong and Gouraud leave silhouettes

Flat
Gouraud
Phong

Comparison (3/3)

- If the polygon mesh approximates surfaces with a high curvatures, Phong smoothing may look smooth when Gouraud shows edges
- Phong smoothing requires much more work than Gouraud smoothing
- Both need data structures to represent meshes so we can obtain vertex normals
- Both leave the silhouette jagged

Flat
Gouraud
Phong

Problems with Interpolated Shading

- Polygonal silhouette
- Perspective distortion
- Orientation dependence
- Unrepresentative surface normals

Foley, van Dam, Feiner, Hughes
Foley, van Dam, Feiner, Hughes

OpenGL Rendering Pipeline

- **Pipeline:** consists of multiple stages. Data flows in, being processed in each stages, then flows out
- **Stages:** each stage represents an unique function to process the input data
 - **Fixed function stages:** limited customization capability, typically exposes states for configuration
 - **Programmable shader stages:** allow custom shader programs to be executed within, providing broader capability of customization

Fixed Function Pipeline (Legacy)

Fixed Function Pipeline (Legacy)

- Before OpenGL 3.0, OpenGL rendering is done in a fixed function pipeline
- Fixed pipeline is like an machine with a lot of switches/values to configure
- One cannot change how the function is implemented as well as the order of execution

Fixed Function Pipeline: Metaphor

OpenGL Fixed Function Pipeline

How do I press these buttons to get desired effect?

Computer Graphics
NTUST CSIE Laboratory

Programmable Pipeline

- Shader programs are introduced in OpenGL 2.0, and included in the core profile in OpenGL 3.0
- Fixed function pipeline is deprecated since OpenGL 3.0
- Shader programs, written in OpenGL Shading Language (GLSL), allow the programmers to customize certain stages in the OpenGL rendering pipeline

Computer Graphics
NTUST CSIE Laboratory

First-Modified Pipeline

- Replace transform and lighting with **vertex shader**
 - Vertex shader must now do transform and lighting
 - But can also do more
- Replace texture stages with **fragment (pixel) shader**
 - Previously, texture stages were only per-pixel operations
 - Fragment shader must do texturing

Computer Graphics
NTUST CSIE Laboratory

The First Generation

- Current hardware allows you to break from the standard illumination model
- Programmable *Vertex Shaders* and *Fragment Shaders* allow you to write a small program that determines how the color of a vertex or pixel is computed
 - Your program has access to the surface normal and position, plus anything else you care to give it (like the light)
 - You can add, subtract, take dot products, and so on
- **Fragment shaders** are most useful for lighting because they operate on every pixel

Computer Graphics
NTUST CSIE Laboratory

The First Generation Example

OpenGL 2.0 Graphics Pipeline

Where can we program?

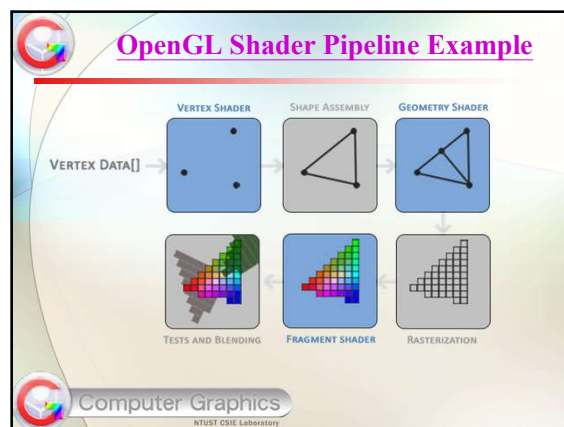
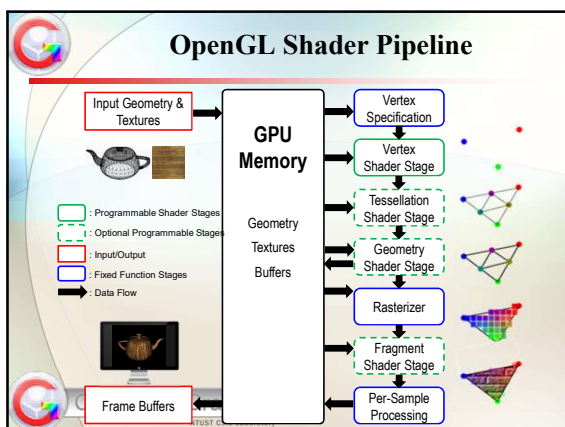
Computer Graphics
NTUST CSIE Laboratory

Programmable Pipeline: Metaphor

Shader Programs

Let's write a program to create the effect!

Computer Graphics
NTUST CSIE Laboratory



Programmable Pipeline V.S. Fixed Function Pipeline

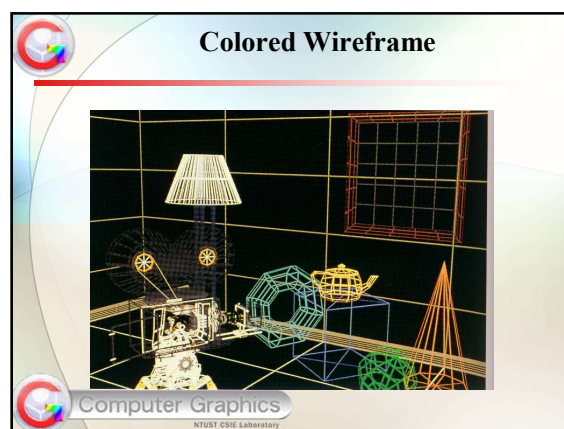
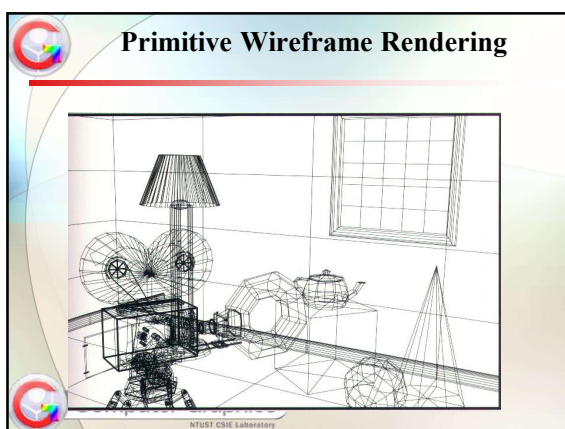
	Programmable Pipeline	Fixed Function Pipeline
Flexibility	+implement various algorithms in shader programs	-limited customization capability
For Simple Application	-must configure the whole pipeline	+performs simple task with less configurations
For Complex Application	+can achieve various effects	-most advanced effects are impossible
Learning Curve	-one must have full knowledge of the pipeline and GLSL before writing an application	+can create simple applications without much knowledge
Deploy	-should consider all graphics driver environment	+works in most graphics driver environment

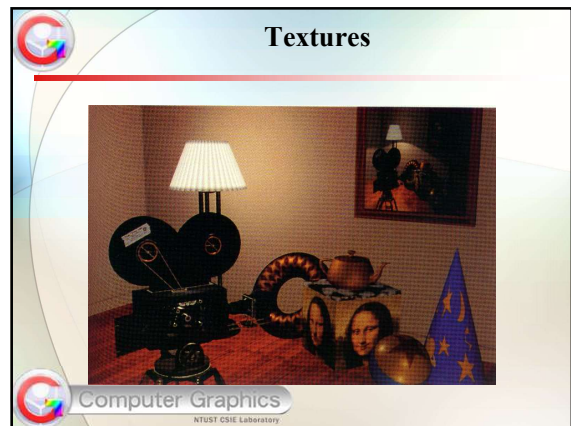
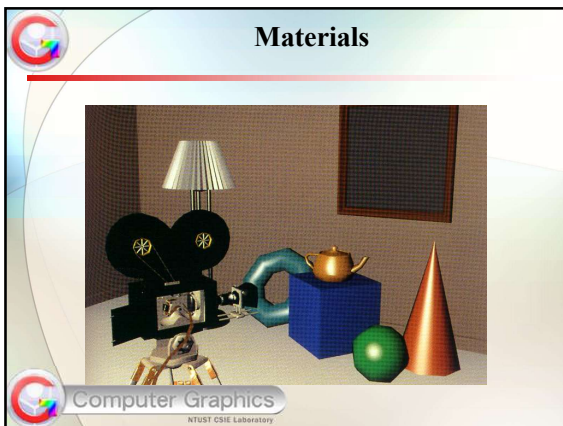
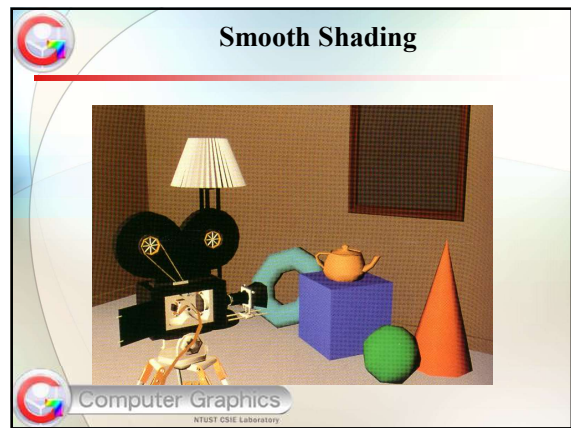
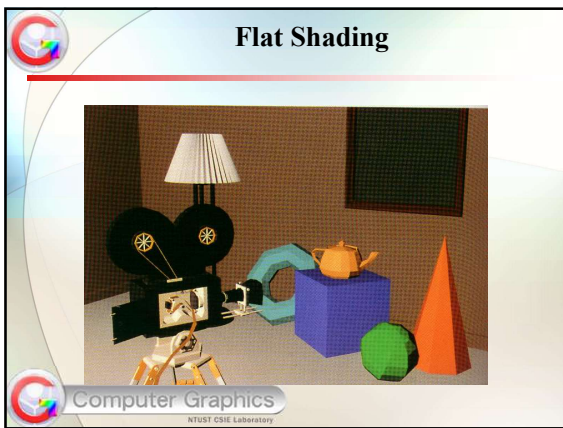
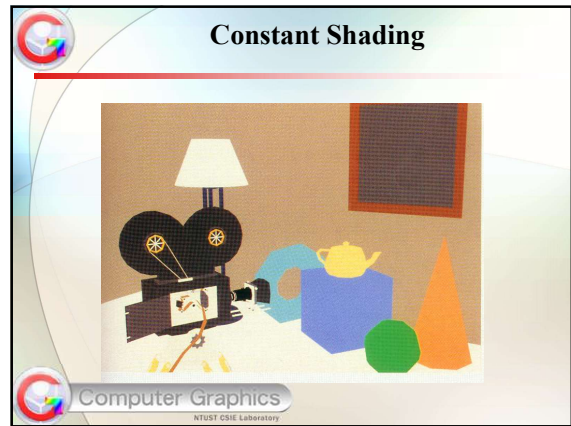
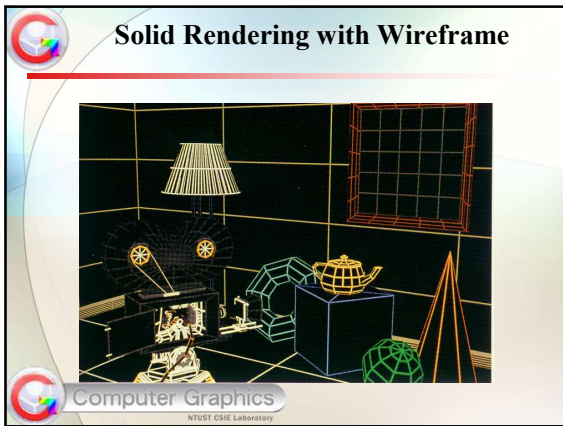
Computer Graphics
 NTUST CSIE Laboratory

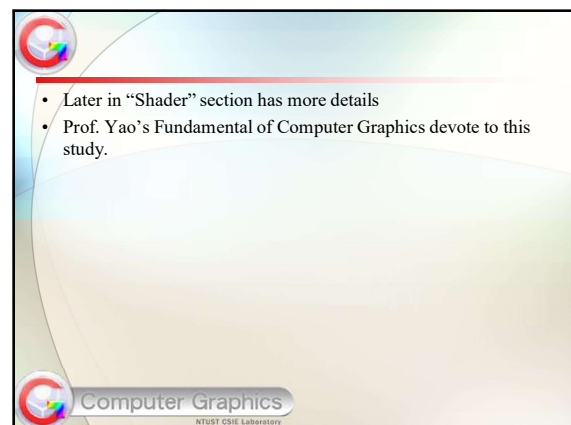
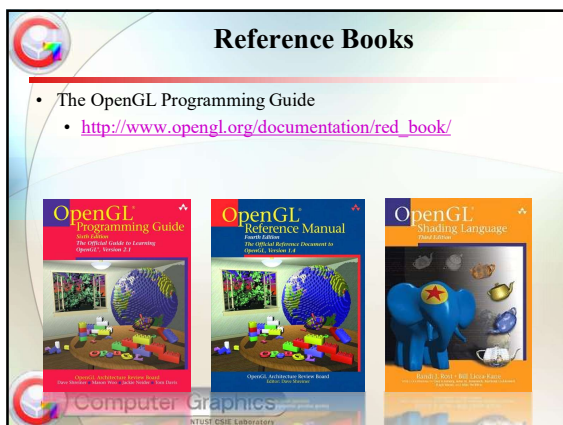
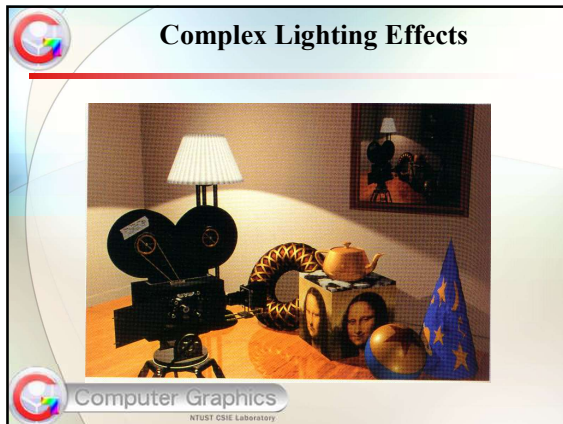
Steps toward reality

A STEP-WISE ILLUSTRATION

Computer Graphics
 NTUST CSIE Laboratory







圖學導論專案



enu

- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 1: An Image Editing Program

Introduction

In this project you will write an image editing program that allows you to load in one or more images and perform various operations on them. Consider it to be a miniature Photoshop.

UI Operations

The operations are summarized here, with details on implementing them below.

Operation	Points
Load	0, provided
Save	0, provided
Difference	0, provided
Run	0, provided
Color to Grayscale	5
Uniform Quantization	5
Popularity	20
Naive Threshold Dithering	3
Brightness Preserving Threshold Dithering	7
Random Dithering	5
Clustered Dithering	10
Floyd-Steinberg Dithering	15
Color Floyd-Steinberg Dithering	10
Box Filter	15 or 3
Bartlett Filter	15 or 3
Gaussian Filter	15 or 3
Arbitrary-Size Gaussian Filter	10
Edge Filter	15 or 3
Enhance Filter	15 or 3
Half Size	8
Double Size	12
Arbitrary Uniform Scale	25 or 10
Arbitrary Rotation	25 or 10
NPR Paint	15 ~ 50

Grading

- In this project you score points for each image operation you correctly implement.
- The possible operations and their values will be listed in the operation section.
- The total number of points up for grabs is greater than 100, but the maximum any individual can get is 100 + 10. Please aim for 110.
- A reference program will be provided so that you can check your implementation, although you may still get full points for an operation even if your result doesn't match the reference program's. Many of the operations are sensitive to very subtle differences in coding, and it is not worth anyone's time to try to have everyone implement everything in exactly the same way. The operation descriptions below indicate the extent to which we think you should match the reference solution.
- We will look for identical programs. If you are found to have duplicated someone else's work you will be treated as a group and given half the earned points. We will also take steps to ensure that you don't submit the reference program and don't manipulate the system in other ways we anticipate.

Submission

Please submit it to the indicate FTP site. TA will send you for further notification

Basic Program

- The basic program must be controllable through a scripting language, and you should not change this in any way. The project will be graded by running scripts, so the scripting interpreter must function. The scripting language is simply a sequence of lines, each of which has a command and some arguments, generally a filename. The commands for each operations are listed below along with the arguments. All arguments should be considered strings. A user should be able to enter a script in a window or load one from a file.
- The program should maintain the current image, which is displayed. The current image is modified by the various operations as outlined below. The skeleton already contains operations which change the current image.
- All files will be in the Targa (tga) format. LibTarga supports pre-multiplied RGBA images. To load the alpha bits, tell it that you are loading 32 bit data, and it will fill the alpha channel (with ones if necessary) along with the color information. When you read an RGBA image with LibTarga, it returns pre-multiplied alpha pixel data. You must divide out the alpha channel before display, taking care to avoid dividing by zero. The skeleton already does this for display.

Program Skeleton

We provide a skeleton programs: [Project1 framework.rar](#). This is for practicing the rendering engine which are kept using in my classes. You should modify the skeleton by changing the file Targalimage.cpp and/or Targalimage.h to implement the functions. At the moment all the functions change the current image to black.

- NOTE: The Targalimage class stores RGBA. Many of the operations only need greyscale, so this is a waste. Ignore it for now by storing grey as RGB with R=G=B, or put separate greyscale image information inside the Targalimage class.
 - NOTE: Many of the operations you need to implement are very similar. For instance, all the filter operations differ only in the filter mask, not the basic filtering algorithm. Write your program to take advantage of such common operations.
 - IMPORTANT: If you choose not to implement a function, it is essential that the function call ClearToBlack and return false. This is the default behavior, so if you don't change it, you should be OK.
 - ALSO IMPORTANT: For each member in your group, you must alter the function MakeNames in Main.cpp so that the function vsStudentNames.push_back is called with the member's name as the argument. We will be using this information during the grading process.
1. A basic program skeleton in ogre version with the scripting language implemented is available. The program provide the proper user interface to do the operations. In addition to the UI. We also provide a script system for you to run a sequence of operations.
 2. A basic program skeleton with the scripting language implemented is available. This program will also load and save images with alpha (if present in the image).
- As it is currently implemented, the skeleton will execute all the commands in a script file that is given as an argument. To specify arguments in Visual Studio, go to the Debug part of the project settings dialog. The skeleton also provides a single line command entry dialog. To execute a

command, type it in and hit the Enter key. Hitting Enter again will run the same command again. You can of course change the command. Try "load test.tga" to load the test image. "save test-save.tga" also works. (Leave out the quotes when you type things in.)

- The skeleton is slightly modular in design. In particular, the widget for displaying an image is separate from the object for storing the image, and both are separate from the main window itself.
- There is a Makefile included in the program skeleton, and the skeleton should compile under Linux. You are welcome to make use of this if you like, but it is an unsupported feature of the project. You should not ask the TAs or the instructor questions about how to make your program run under Linux.

Supporting Programs

We provide the following programs:

- A reference program that implements all the operations.
- We do not support you doing this project under Linux, however, there is a Linux binary reference program that you can use in this unsupported capacity.
- A program that shows targa files along with their alpha channel. Use this to test your compositing operations.

We also provide a whole range of example images, some with non trivial alpha channels.

Details on Things to Implement

Things in bold are category headings. The comments associated with each category apply to all the sub-operations. For instance, the comments associated with Filtering apply to all of the filtering operations.

Operation	Keyword	Arguments	Details	Points
Load	load	filename	Load the specified image file and make it the current image.	0, provided
Save	save	filename	Save the current image to the specified file.	0, provided
Difference	diff	filename	Subtract the given image file from the current image and put the result in the current image.	0, provided
Run	run	filename	Executes the script named filename. The script should contain a sequence of other commands for the program, one per line. The script must end with a newline.	0, provided
Color to Grayscale	gray		Use the formula $I = 0.299r + 0.587g + 0.114b$ to convert color images to grayscale. This will be a key pre-requisite for many other operations. This operation should not affect alpha in any way.	5
24 to 8 bit Color			All of these operations assume that the current image has 24 bits of color information. They should still produce 24 bit images, but there should only be 256 different colors in the resulting image (so the image could be stored as an 8 bit indexed color image). Don't be concerned with what happens if you run these operations on something that is already quantized. These operations should not affect alpha - we will only test them on images with alpha = 1 (fully opaque images).	
Uniform Quantization	quant-unif		Use the uniform quantization algorithm to convert the current image from a 24 bit color image to an 8 bit color image. Use 4 shades of blue, 8 shades of red, and 8 shades of green in the quantized image.	5
Populosity	quant-pop		Use the populosity algorithm to convert the current 24 bit color image to an 8 bit color image. Before building the color usage histogram, do a uniform quantization step down to 32 shades of each primary. This gives $32 \times 32 \times 32 = 32768$ possible colors. Then find the 256 most popular colors, then map the original colors onto their closest chosen color. To	20

find the closest color, use the euclidean (L2) distance in RGB space. If (r_1, g_1, b_1) and (r_2, g_2, b_2) are the colors, use $\sqrt{(r_1-r_2)^2 + (g_1-g_2)^2 + (b_1-b_2)^2}$ suitably converted into C++ code.

Dithering			All of these operations should convert the current image into an image that only contains black and white pixels, with the exception of dither-color. If the current image is color, you should first convert it to grayscale in the range 0 - 1 (in fact, you could convert all images to grayscale - it won't hurt already gray images). We will only test these operations on images with alpha = 1.	
Naive Threshold Dithering	dither-thresh		Dither an image to black and white using threshold dithering with a threshold of 0.5.	3
Brightness Preserving Threshold Dithering	dither-bright		Dither an image to black and white using threshold dithering with a threshold chosen to keep the average brightness constant.	7
Random Dithering	dither- rand		Dither an image to black and white using random dithering. Add random values chosen uniformly from the range $[-0.2, 0.2]$, assuming that the input image intensity runs from 0 to 1 (scale appropriately). There is no easy way to match the reference program with this method, so do not try. Use either a threshold of 0.5 or the brightness preserving threshold - your choice.	5
Clustered Dithering	dither- cluster		Dither an image to black and white using cluster dithering with the matrix shown below. The image pixels should be compared to a threshold that depends on the dither matrix below. The pixel should be drawn white if: $I[x][y] \geq \text{mask}[x\%4][y\%4]$. The matrix is: $\begin{matrix} 0.7059 & 0.3529 & 0.5882 & 0.2353 & 0.0588 & 0.9412 \\ 0.8235 & 0.4118 & 0.4706 & 0.7647 & 0.8824 & 0.1176 \\ 0.1765 & 0.5294 & 0.2941 & 0.6471 & & \end{matrix}$	10
Floyd-Steinberg Dithering	dither- fs		Dither an image to black and white using Floyd-Steinberg dithering as described in class. (Distribution of error to four neighbors and zig-zag ordering).	15
Color Floyd-Steinberg Dithering	dither- color		Dither an image to 8 bit color using Floyd-Steinberg dithering as described in class. You should use the color table corresponding to uniform quantization. That is, the table containing all colors with a red value of 0, 36, 73, 109, 146, 182, 219 or 255, green in the same range, and blue in the set 0, 85, 170, 255. If you do this, but not the grayscale version of Floyd-Steinberg, then you get 15 points.	10
Filtering			All of these operations should modify the current image, and assume color images. The alpha channel should NOT be filtered. The alpha channel for all the test images will be 1 for all pixels, so you do not need to worry about the differences between filtering regular pixels or pre-multiplied pixels. Implement whichever approach you prefer.	15 for the first 3 for any additional
Box Filter	filter- box		Apply a 5x5 box filter.	
Bartlett Filter	filter- bartlett		Apply a 5x5 Bartlett filter.	
Gaussian Filter	filter- gauss		Apply a 5x5 Gaussian filter.	
Arbitrary-Size Gaussian Filter	filter- gauss- n	N (size)	Apply an NxN Gaussian filter. Use the binomial method presented in lecture to derive the filter values. Note that this is the same Gaussian you will use if you do the NPR paint task.	10

(circular) brush size version from section 2.1 of this paper. A function to do the actual drawing of the circular strokes (`TargalImage::Paint_Stroke`) has been provided for you.

To match the reference solution (which is what you're graded on), your implementation should use the brush size radii of 7, 3 and 1. When calling the Gaussian-blur function, use the filter constructed using the binomial coefficients with a filter size of

$$2 \times \text{radius} + 1$$

The `fg` parameter should be set to 1, and the threshold parameter `T` should be set to 25.

The difference function in Hertzmann's pseudo-code is simply Euclidean distance (as specified in the text below the `paintLayer` figure), so you'll need to compute and store these distances on a per-pixel basis.

- Advance (15 ~ 50)

You can add stroke or other effects into NPR rendering and your score depends on how impressive your work.

Sample Results

Sample Results : [Solution.rar](#)

You can use the reference program to generate sample images, and then use the difference operation to compare your results with the sample. The table below summarizes ways in which your results could reasonably differ from the reference program's.

Operation	Test Images(s)	Notes
gray	colors-for-bw.tga	You should be able to reproduce this exactly.
quant-unif	church.tga and wiz.tga	You probably cannot re-produce this exactly. Your result should, however, show the same poor quality and color banding effects.
quant-popul	church.tga and wiz.tga	You probably cannot re-produce this exactly. A populosity algorithm should do a reasonable job on the gray floor, and not too bad on the browns. It should, however, draw the blue ball as gray, because there are not enough blue pixels to be popular.
dither-thresh	church.tga	You should be able to reproduce this almost exactly. Some pixels may be different around the boundaries between white and black.
dither-bright	church.tga	You should be able to reproduce this almost exactly. Some pixels may be different around the boundaries between white and black.
dither-rand	church.tga	You have no chance of reproducing this exactly. Instead, you should get an image that is similar in style but not identical.
dither-order	church.tga	You should be able to reproduce this almost exactly. A few borderline pixels (those close to the threshold) may be different.
dither-cluster	church.tga	You should be able to reproduce this almost exactly. A few borderline pixels (those close to the threshold) may be different.
dither-fs	church.tga	There's a good chance you can re-produce this exactly, but it is not essential. The character of your result should be similar.
dither-color	church.tga	There's a good chance you can re-produce this exactly, but it is not essential. The character of your result should be similar.
filter-box	church.tga and checker.tga	You may get different results around the boundary, but interior pixels should be identical. The reference program extended the size of the input image by reflecting it about its edges.
filter-	church.tga and checker	You may get different results around the boundary, but interior

bartlett		pixels should be identical. The reference program extended the size of the input image by reflecting it about its edges.
filter-gauss	church.tga and checker	You may get different results around the boundary, but interior pixels should be identical. The reference program extended the size of the input image by reflecting it about its edges.
filter-gauss-n	church.tga and checker	The differences are the same for the 5x5 version of the gaussian filter.
half	church.tga and checker.tga	You may get slightly different results, particularly around the boundary.
double	church-small.tga and checkers-small.tga	You may get slightly different results, particularly around the boundary.
scale	church.tga and checker.tga	You may get different results, but they should be qualitatively similar (no banding).
rotate	church.tga and checker.tga	You may get slightly different results, particularly around the boundary.
npr-paint	church.tga and wiz.tga	This is a randomized algorithm, so it is very unlikely that your results will match the reference solution exactly (the reference solution operating twice on the same image is unlikely to match itself exactly). Your results should be qualitatively similar to the output of the reference solution, but need not be pixel-wise identical.

Copyright © 2019 NTUST CSIE Computer Graphics Lab. All right reserved.



Menu

- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 2: Maze Visibility and Rendering Graphics

Introduction

Your task in this project is to implement a maze rendering program, not too far removed from those used in computer games of the first-person variety. Read this entire document carefully before beginning, as it provides details of the required implementation and various tips.

Mazes

A maze consists of rectangular cells separated by edges. The edges may be either transparent or opaque. The viewer is supposed to see through transparent edges into the neighboring cells, and they should not see through opaque edges. Each edge is assigned a color (which is meaningless for transparent edges).

The maze is described as a 2D structure assumed to lie in the XY plane. To make it 3D, each edge is extruded vertically from the floor to the ceiling. The floor is at $z=-1$ and the ceiling is at $z=1$. Each wall should be drawn with its assigned color.

Associated with the maze is a viewer. The viewer has an (x,y,z) location, a viewing direction, and a horizontal field of view. The view direction is measured in degrees of rotation counter-clockwise about the positive z axis. The horizontal field of view is also measured in degrees. For the project, the viewer's z will always be 0.

The maze file format consists of the following information (also look at one of the example mazes):

- The number of vertices in the maze, nv . Each edge joins two vertices.
- The location of each vertex, specified as its x and y coordinates. The vertices are assumed to be numbered from 0 to $nv - 1$.
- The number of edges in the maze, ne . Remember, there is an edge between every cell, even if that edge is transparent.
- The data for each edge: the index of its start vertex, the index of its end vertex, the index of the cell to the left, the index of the cell to the right, a 1 if the edge is opaque, or 0 if transparent, and an RGB triple for the color. The left side of an edge is the side that would appear to be on your left if you stood at the start of the edge and looked toward its end. If there is no cell to the left or right, an index of -1 is used. The edges are assumed to be numbered from 0 to $ne - 1$.
- The number of cells in the maze, nc .
- The data for each cell, which consists of the four indices for the edges of the cell. The indices are given in counter-clockwise order around the cell.
- The view data, consisting of the (x, y, z) viewer location, viewing direction and the horizontal field of view.

Software Provided

Source code can be found in the [MazeFramework.rar](#). Example Executable code is in [exe](#).

Several classes have been provided. Together they build two programs. The first program creates mazes in a certain format. The second is a skeleton maze renderer. The code is reasonably well

documented, but part of the project is figuring out how the given code works and how to integrate your code into it. The programs are described below. To build them, set the appropriate startup project in Visual Studio and build.

- **BuildMaze**

The BuildMaze program provides a simple user interface for building mazes. The user specifies the following parameters:

Cells in X: The number of cells in the x direction. Cells in Y: The number of cells in the y direction. Cell X Size: The size of the cells in the x direction. Cell Y Size: The size of the cells in the y direction. Viewer X: The initial x location of the viewer. Viewer Y: The initial y location of the viewer. Viewer Z: The initial z location of the viewer. Viewer Dir: The initial viewing direction, given in degrees of rotation about the positive z axis (the standard way of specifying a rotation in the plane). Viewer FOV: The horizontal field of view of the viewer.

The Build Maze button builds a maze with the given parameters and displays it. The Save Maze button requests a file name then saves the maze to that file. The Load Maze button requests a maze file to load and display. Quit should be obvious.

- **RunMaze**

The RunMaze program provides a skeleton for the maze walkthrough that you will implement. As provided, it displays both a map of the maze and an OpenGL window in which to render the maze from the viewer's point of view. On the map is a red frustum indicating the current viewer location, viewing direction and field of view. The map is intended to help you debug your program by indicating what the viewer should be able to see.

To move the viewer, hold down a W/S/A/D and Left/Right Arrow in the OpenGL window. Key W or Key S is translated as forward or reverse motion of the viewer. Key A and Key D is translated as move Left and Right. Left and Right Arrow motion changes the direction of view. As the skeleton exists now, the viewer will move in the map window to reflect the Keyboard Control.

The system performs collision detection between the wall and the viewer to prevent the viewer from passing through opaque walls. You should examine the code that does that to see an implementation of clipping that clips a line segment to an edge using an approach similar to Liang-Barsky clipping. The RunMaze program "do not" keeps track of which cell the viewer is currently in, which is essential information for the cell-portal visibility algorithm you must implement.

You should pay particular attention to the function Mini_Map in OpenGLWidget.cpp that sets up the OpenGL context for the window. As you will read later, all of the drawing you do in this project must be in 2D, so the window is set up as an orthogonal projection using the special OpenGL utility function gluOrtho2D. The Mini_Map function also draws the projection of the ceiling and the floor of the maze. You should be able to reason as to why is it safe to treat the floor and ceiling as infinite planes (hint: the maze is closed), and why those planes project to two rectangles covering the bottom and top half of the window. You do not need to change this function.

- **C++ Classes**

This document will not go into details of the C++ classes provided. You should spend a considerable amount of time perusing them to figure out how everything works, and too look for little functions that will be useful in your implementation, such as functions to convert degrees to radians and back again (recall that all the C++ trigonometry functions take radians).

Your Task

Produce the viewer's view of the maze. You must extend the function OpenGLWidget::Map_3D to draw what the viewer would see given the maze and the current viewing parameters. Note that the function is passed the focal distance, and you also have access to the horizontal field of view. Your implementation must have the following properties.

- You must use the Cell and Portal visibility algorithm to achieve exact visibility. In other words, apart from drawing over the floor and ceiling, no pixel should be drawn more than once. The algorithm is given in pseudocode below.

```

Draw_Cell(cell C frustum F){
    for each cell edge E{
        if E is opaque{
            E' = clip E to F
            draw E'
        }
        if E is transparent
            E' = clip E to F
            F' = F restricted to E'
            Draw_Cell(neighbor(C,E),F')
        }
    }
}

```

Note that you should implement the following functions. The function `Draw_Cell(C, F)` is initially called with the cell containing the viewer, and the full view frustum. The `neighbor(C,E)` function returns the cell's neighbor across the edge. Note that drawing a 2D edge means drawing a wall in 3D.

- You are only allowed to use OpenGL 2D drawing commands. In other words, any vertices you specify should use `glVertex2f`, `glVertex2fv`, `glVertex2d` or `glVertex2dv` only. You should use `glBegin(GL_POLYGON)` to draw polygons, and `glColor3f` or `glColor3fv` to specify the polygon color. We will check for other OpenGL calls when we grade.
- As an side effect of the 2D restriction, you must do your own viewing transformation. That is, you must take points specified in world space (where you will do the visibility) and transform them all the way into screen space (where you will draw them.) The transformation will consist of a translation and rotation to take the points from world to view space (with the origin at the viewer's location) and then a perspective division to take the view space points into screen space. Note that you are given the focal distance to make things easier, but you must still take care of several small details. Note that you can do all the transformations using basic transformation matrices followed by simple perspective. You DO NOT need to construct general purpose viewing transformation matrices.

Helpful Tips

- The visibility algorithm is a 2D algorithm in this case, because all the walls are vertical and the viewer is looking horizontally. That also means that all the pieces of wall that you draw will have vertical left and right edges. They will not have horizontal top and bottom edges due to perspective effects.
- Implement a Frustum class that stores information about a viewing frustum, and has a method for clipping a frustum to an edge. The easiest way to represent a frustum is as a point for the viewer's location, and two edges for the left and right "clipping lines" of the frustum. There is no near and far clip lines in this project.
- Implement a function in the Edge class or the LineSeg class that clips an edge to a given view frustum. You will have to work out a way to compute the intersection point of a line segment with an infinite line in 2D space. Start by writing out the equations of the lines in parametric coordinates. There is a function in the LineSeg class that may help get you started.
- It is easiest to begin with a 1 by 1 maze, in which case there is no recursive step. That gives you the opportunity to debug the transformations and projection before getting into the details of manipulating view frustums.
- To do the projection, first translate the point so that the viewer is at the origin (subtract the viewer's location from the point.) Then undo the viewer's rotation direction by rotating the point. Then do the perspective divide (using the simple perspective projection from the notes, suitably modified.)
- Note that in OpenGL 2D drawing, x is to the right and y is up. When you do the transformation above, you end up with y to the left, z up and x into the screen. You have to fix this problem.

It cannot be stressed enough: This project can be completed in somewhere between one hundred and five hundred lines of code. Spend a lot of time thinking about what you are trying to do with each piece of code. And spend a lot of time looking at the code you are given. Start with pen and paper, because there are a lot of small pieces of math that you need to work out. Grading and Submission

Grade

The project will be graded out of 50. You get:

- 20 points for projecting a wall onto the screen with perspective projection and drawing it.
- 10 more points for correctly clipping walls to the view, for a single cell maze.
- 20 more points for doing the recursive visibility.

Submission

Submission will work similar to project 1. Grading of this project will be by demo in a series of face-to-face grading sessions.

Criteria

You must work alone for this project.

Copyright © 2019 NTUST CSIE Computer Graphics Lab. All right reserved.



enu

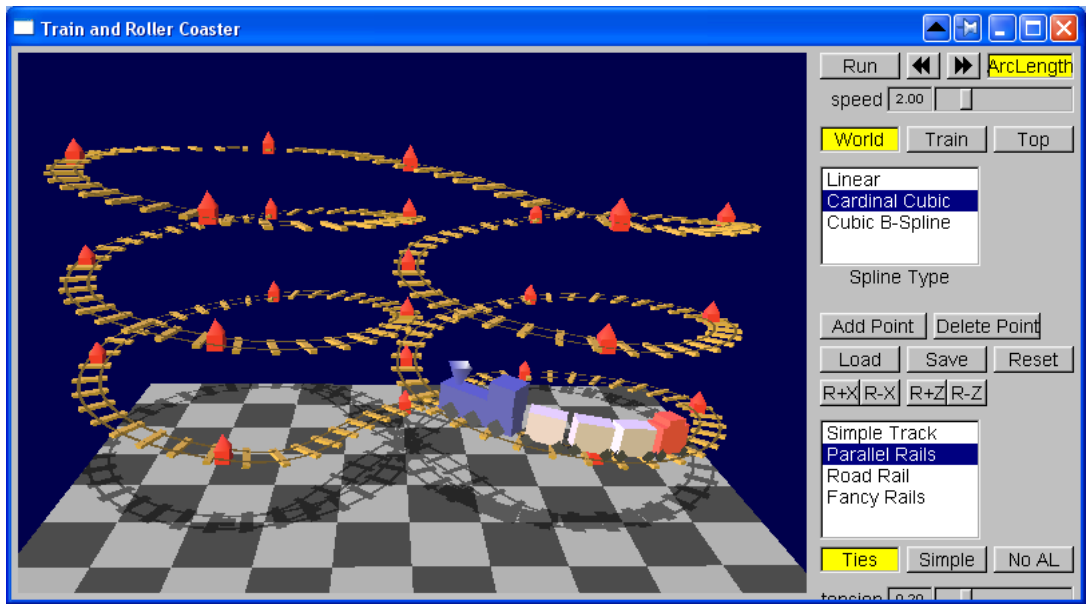
- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 3: A Tiny Amusement Park with Roller Coasters and Water Shaders

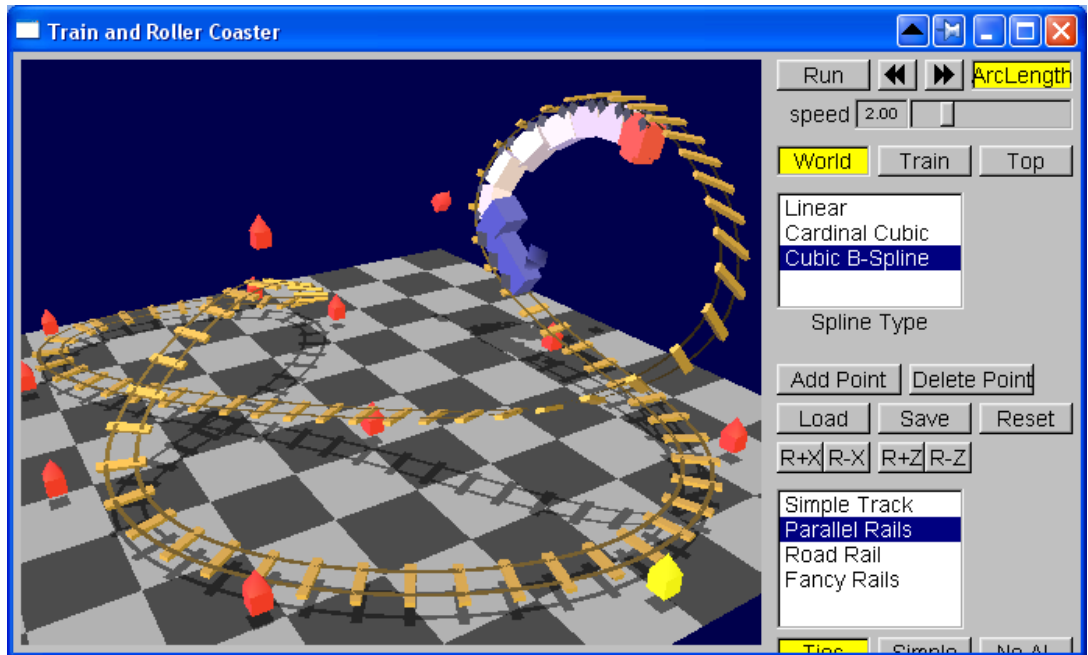
Overview

This project would like you to create an amusement park with roller coasters allowing users to have fun and interact with the program. Inside the amusement park, you must also explore a few possibility to simulate the water surface and render the surface with shader programs which become more and more important in graphics. At the end, you can use your creativity to create an interesting environment or an interactive simple game.

In this project, you will create a train that will ride around on a track. When the track leaves the ground (or is very hilly), the train becomes more like a roller coaster.



Once it becomes a roller coaster, loops, corkscrews, and other things become possible



The main purposes of this project is to give you experience in working with curves (e.g. the train and roller coaster tracks). It will also force you to think about coordinate systems (to make sure that things move around the track correctly). Thus, we will provide you framework code so that you don't need to worry about that so much.

The core of the project is a program that creates a 3D world, and to allow the user to place a train (or roller coaster) track in the world. This means that the user needs to be able to see and manipulate a set of control points that define the curve that is the track, and that you can draw the track and animate the train moving along the track. We'll provide the framework code that has a world and manages a set of control points. You need to draw a track through those points, put a train on that track, and have the train move along the track.

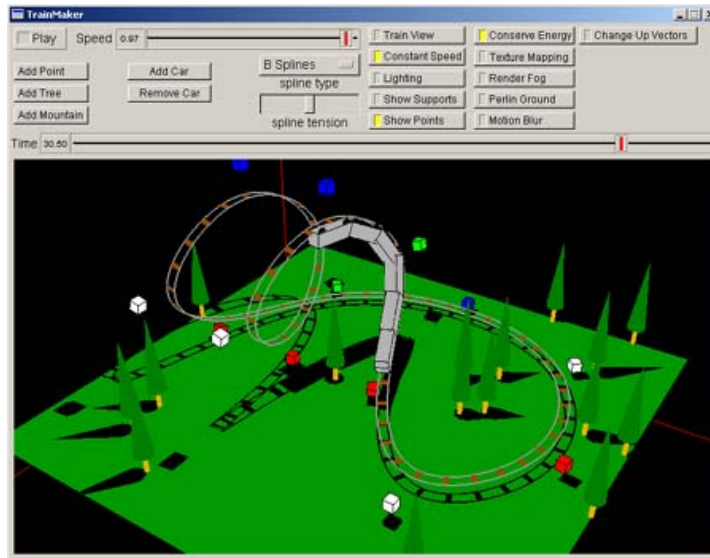
Basically in this project you will need to:

- Find your way around the framework code.
- Add the basic functionality: draw a track (curve) based on the control points and draw a train on that track. If you do the latter part correctly, the framework will make it easy to animate the train going around the track. You will also need to implement a "train view" (so the user can "ride" your train).
- Add more advanced features: nicer drawing of the track, arc-length parameterization, more kinds of splines, physics, ...
- Add special effects and extra features to make it really fun. Really nice looking train cars, scenery, better interfaces for creating complex tracks, ...

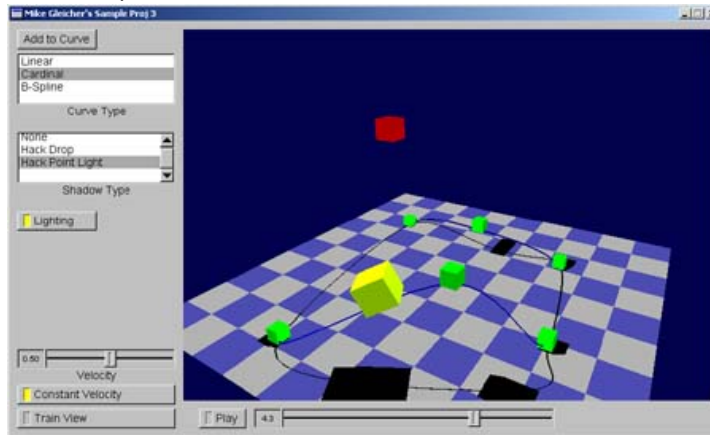
I must emphasize that the basic functionality is most important, and the core advanced features (arc-length parameterization) are the next most important things. Fancy appearance (like using textures and pretty lighting) aren't the focus here - add them only if you have time after doing the more important things.

We have provided a [sample solution](#) of the possible features (at least the most common ones). We recommend that you play with it a bit to understand how it works. The example also has options that lets you see some of the most common mistakes and simplifications that students make.

While the assignment was a little bit different in 1999, the basic idea was the same. For a totally crazy solution to this assignment, check out [RocketCoaster](#). It was what happened when I let two students work as a team. There are two more "normal" example solutions to this project which are by and [Rob](#), both from 1999. One is a version that I wrote (called mikes-Train) and another was written by a really good student (robs-train). I recommend that you try them out to get an idea as to what you'll be doing.

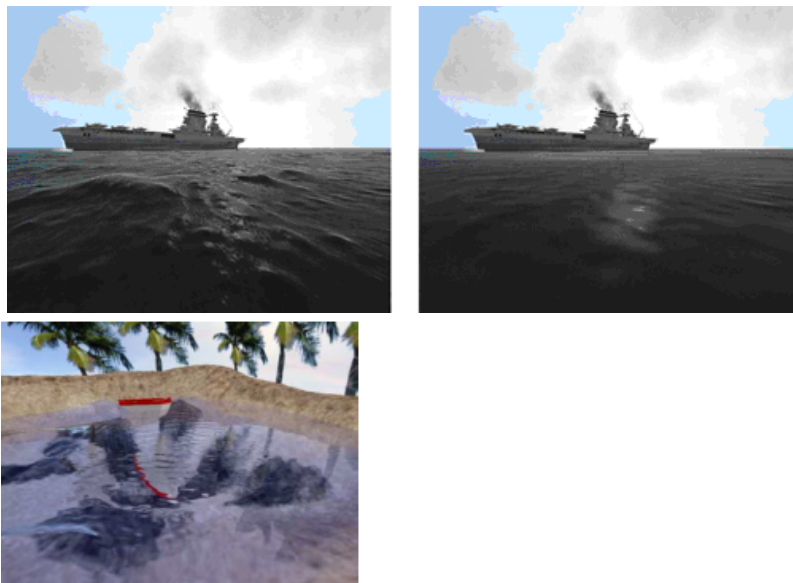


Mike's sample train, circa 1999



Rob Iverson's A+ assignment from 1999

Water Introduction



Water is commonly seen in our daily life. However, the interaction and rendering of water involve complex physical computation. Furthermore, interactive graphics is a program which can interact with the virtual world created by you.

Water Surface Generation With GPU

Basically, the water surface can be simply decomposed into two geometric components: high-level surface structure and low-level surface details.

- The surface structure represent the large scales of the wave movement and can be represented with a set of 2D grid with the y direction represent the height of the grid point. A vertex shader can be used to simulate the movement at the vertices by changing the normal and the y position of the vertex.
- The surface detail represent the small scale perturbation in the local area and generally can be represented as a normal map. A pixel shader is created to generate a normal map for those perturbation.

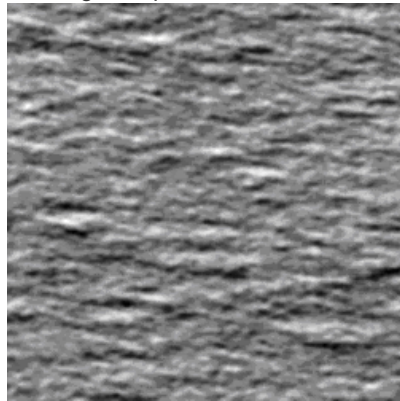
Generally, there are several ways to implement the wave on the surface of water:

- Sine waves (10) (Ref. 1): The sum of the sine waves are chosen to represent the complex water surface movement. The equation can be expressed as

$$W_i(x, y, t) = A_i \times \sin(\mathbf{D}_i \cdot (x, y) \times w_i + t \times \varphi_i).$$

$$H(x, y, t) = \sum (A_i \times \sin(\mathbf{D}_i \cdot (x, y) \times w_i + t \times \varphi_i)),$$

- Wavelength (L): the crest-to-crest distance between waves in world space. Wavelength L relates to frequency w as $w = 2\pi/L$.
- Amplitude (A): the height from the water plane to the wave crest.
- Speed (S): the distance the crest moves forward per second. It is convenient to express speed as phase-constant, phi, where $\phi = S \times 2\pi/L$.
- Direction (D): the horizontal vector perpendicular to the wave front along which the crest travels. Please refer to Ref. 1 for more direction details.
- Height maps (10) (Ref. 3): similar to sine wave method, height map method decomposes the wave on the water surfaces into a set of different level of detail represented as a heightmap (the shape of single component and generated by artists) The following shows an example of the height map.



The combination of the heightmap can be expressed as the following equation.

$$H(x, y, t) = \sum_n^N b(A_i^x x + B_i^x, A_i^y y + B_i^y, A_i^t t + B_i^t).$$

Please refer to Ref. 3 for more heightmap details.

- Wave equation(Ref. 4): Generally, the movement of the wave can be expressed as a wave equation as listed in the following:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2} \right)$$

Then by modeling the water surface as cubic Bspline surfaces, the right hand side of the equation can be expressed by the following:

$$c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2} \right) = c^2 ([y_{-1,0} + y_{1,0} - 2 \cdot y_{0,0}] + [y_{0,-1} + y_{0,1} - 2])$$

$$c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2} \right) = c^2 (y_{-1,0} + y_{1,0} + y_{0,-1} + y_{0,1} - 4 \cdot y_{0,0})$$

And then the integration can be computed with one of the following integration methods

$$p(t_2) = 2 \cdot p(t_1) - p(t_0) + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

$$p(t_2) = 2 \cdot p(t_1) - p(t_0) + a(t_1) \cdot h^2$$

$$p(t_2) = (1 + \alpha) \cdot p(t_1) + \alpha \cdot p(t_0) + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

Then the equation is solved at each vertex of the 2D mesh to generate the water surface. Please refer to Ref. 4 for details.

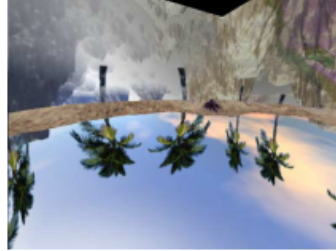
Water Surface Rendering With GPU

Two important effects for the water surface rendering: reflection and refraction. We assume that we are looking outside the water. The reflection and refraction can be generated with the following different methods

- Combination of refraction and reflection with an environment map for the sky and a set of texture map for the box. (Ref. 1) (10)
- Using the refraction map and reflection map to simulate the possible viewing condition from above the water to simulate the refraction and reflection effects. (Ref. 4) (10). The following shows an example of refraction and reflection map

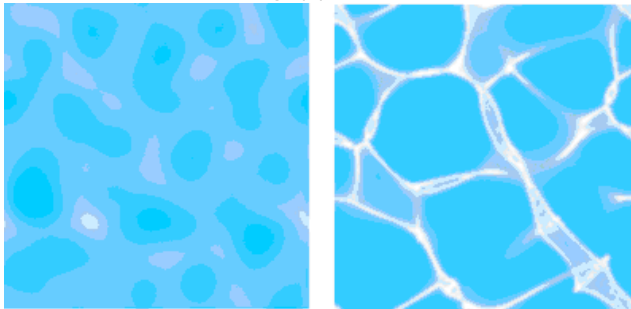


Refraction Map



Reflection Map

In addition to the refraction and reflection, the caustics on the floor due to concentration of refraction and reflection is also visually important. Therefore, simulate the caustics effect on the floor is also important for the rendering. (5)



Ground Rules

You can complete the assignment individually or in pair. We allow (encourage) you to discuss the project with your classmates, but the things you turn in must be substantively your own.

Your program must use OpenGL or OGRE3D and run on the computers in RB-508 (like everything else in this class). While we strongly recommend you use the framework code, this is not a requirement. The Framework/Example solution uses FITk or OGRE3D, but you can use any UI toolkit that you wish, providing its available in the Lab (talk to us) as we need to be able to build your program.

If you want to use other external libraries/code, please ask us. Something like a math library or data structures library is probably OK, but please check.

The Basic Functionality and Framework and Components

The most basic part of this assignment is to provide a "track" for the train track. Your program must do the following things (we provided three version of framework: the [\):Project3Framework.rar](#)

- Provide a user interface to look around the world, as well as providing a "top down" view.
- You must have a "ground" (so the track isn't just in space).
- Provide a user interface that allows control points to be added, removed, or repositioned. Note: even if you do a very advanced interface, you should display the control points and allow for them to be edited manually.
- Allow for the control points to be saved and loaded from text files in the format used by the example solution.
- Provide lighting.

- Allow things to be animated (have a switch that allows the train to start/stop), as well as allowing for manually moving the train forward and backwards.

If you make your own framework, please make sure you can do all of these things. You don't get any extra points for writing it yourself, but you will lose points if you don't have the basic features.

The basic/essential features and components you must add:

Roller Coasters:

- Have a CubicBSpline track. Your program should draw the track. The track should be a loop, always, and should be either interpolate or approximate the control points.
- Have a train that goes around the track (with a play button to start/stop it). The train should always be on the track. Your train need not be fancy, but it should be obvious which end is the front. And your train should not distort in wierd ways as it moves (if it is not rigid, it should be for a good reason).
- Have the train oriented correctly on the track. The train should always face forward if the track is flat, and mostly face forward on a 3D track. Getting 3D orientation correct in the hard cases (like loops) is a more advanced feature (see below).
- Allow the user to "ride" the train (look out from the front of the train). There should be a button or keystroke to switch to this view.
- Have some scenery in the world besides the groundplane.
- Your program is properly documented, is turned in correctly, and has sufficient instructions on how to use it in the readme file.
- You should have a slider (or some control) that allows for the speed of the train to be adjusted (how far the train goes on each step, not the number of steps per second).

Amusement Part:

- at least a water surface with interaction
- water rendering effects: refraction and reflection
- Multiple objects moving at any time (besides the ones that I made)
- Multiple different types of behaviors (besides the ones that I made)
- Multiple different types of buildings / scenery (besides the ones that I made)
- Multiple new textures. Some must be hand painted. Some must not be flat (that is, it must wrap onto multiple polygons)
- You must attempt "enough" technical challenges (see the technical challenges page).
- You must have at least 3 shaders in your program (by "shader" we mean a pair of vertex/fragment programs attached to an object). At least one of these shaders must provide a procedural texture, and at least one of the shaders must be (properly) affected by the lighting. At least one of the shaders must be affected by the time of day (so you need to figure out how to pass the time of day to the shader).
- You program must work at a sufficient frame rate (which isn't hard since the tomputers are so fast).
- You must add something that is effected by the time of day (besides the one shader used to fullfill the requirement above). For example, you can have an object that changes color (the shader is sensitive to the time of day) and shape (something besides the shader is sensitive to the time of day).
- You must use at least one type of "advanced" texture mapping: multi-texturing, projective (slide projector) texturing, environment mapping, bump mapping, or shadow mapping. (if you want to pick something not on this list, you may want to check with us to make sure it counts) 10. An object made out of a curved surface. You can implement subdivision, or some form of parametric surfaces, or do a surface of revolution, or ... This is described more on the technical challenges page.

The framework code is designed to make it easy to add all of those things. In fact, there are "TODO:" comments explaining where to plug them in. See the the discussion of it here.

The framework code was used to make the sample solution. We didn't give you all of the files, but you can see the "hooks" to the parts we didn't give you (they are turned off with a macro). In some places, we intentionally left extra code for you to look at as a hint. The framework has some spiffier features (like drop shadows), and some features you may not need (the control points have "orientation").

With the advance of GPU technology, GPU becomes more and more popular in computer graphics specially in game industry. It is also noticed by general computation community. This project will provide you with experience at programming shaders with GPUs, modeling objects and creating interactive animations for computer graphics, and introduce you to many more of the features of OGRE3D. Your goal is to make the scene as a scene park based on the roller coaster. The overall goal of this project is to give you an opportunity to explore topics in interactive graphics: how do you make things that look interesting, and be interactive. While some of this is artistic (you need to pick interesting objects to make and good textures/... to look nice), some of it is technical: you need to pick things that can be implemented efficiently and have interesting behavior. Like project 1, this project defines a set of sub-goals with points awarded for each goal. Unlike project 1, the goals are far more loosely defined, so there is scope to try interesting things to get all the points available.

Furthermore, water surfaces are common in computer graphics, especially in games for creating fantastic effects. They are a critical element that can significantly improve the level of realism in a scene. But depicting them realistically is a hard problem, because of the high visual complexity present in the motion of water surfaces, as well as in the way light interacts with water. This project would give you the chance to explore techniques developed for rendering realistic depictions of the water surface. With these two things in mind, this project would like you to have the experience by implementing a shader program and a CUDA program to generate a water surface and then rendering it with reflection and refraction effects. In addition, ray-tracing is very important global illumination algorithm to generate realistic images in graphics community. Therefore, the project also ask you to implement a ray tracer for rendering the water and other scene objects.

In terms of your grade, effort spent on technical are more valuable because we are computer scientist. For example, it is better to spend your time making a simple "blocky" car drive around in an interesting way, or to make a simple shaped car out of parametric surfaces, or to light the car in an interesting way, then to carefully model a gorgeous model of a car. (of course, if you want to make model a gorgeous car, implement bezier patches to display its curved body, have it realistically race around a track ... - we won't complain).

Some specific things we want you to learn from this assignment (which will explain some of the requirements):

1. To try out some of the technical topics that we've discussed in class (subdivision surfaces, culling, ...) or topics we won't discuss too much in class (particle systems, fractals, ...)
2. To get some experience with how textures are used to make simple objects look more interesting.
3. To get some experience with creating geometry for graphics.
4. To gain experience working with a larger, more complex graphics application.
5. To gain some experience creating the behavior/motion of graphics objects.
6. To work with shaders for generating water surfaces and shading effects

The Tasks

Fundamentally, to generate the perception of the water in an interactive application consists of two tasks:

1. Your program has to generate a water surface according to some physical rules which are formed for some specific phenomenon in your mind. This generated surface represents the boundary of water for a renderer to generate proper images for illustrating the water surface.
2. Your program must render the water surface to simulate the effects of refraction and reflection. Furthermore, the caustics caused by water movement on the sea floor can be added to add more reality.

In addition, realistically rendering the water surface is also important for other graphics applications, a basic ray-tracing algorithm is required to simulate the global illumination effect in the scene. The ray tracer will generate an image according to rays through the center of all pixels to interact with the scene object and then compute the lighting effects and shadow effects.

Furthermore, each task requires modeling one or more objects using a specific technique from class. The points available for each technique varies according to the difficulty of the task. In all cases, you get a base number of points for implementing one object with a technique, then an extra 5 points for each additional, but distinct, object with the same technique. You can score points for a maximum of three objects with any one technique. For instance, if you create a texture mapped polygonal ticket booth, and a texture-mapped polygonal roller-coaster carriage, and extrude the roller-coaster tracks, then you get $20 + 5 + 25 = 50$ points. If an object involves more than one thing, such as a texture mapped, swept surface, then you can score points for both texture mapping and sweep objects. The maximum number of points is 100. As with Project 1, you can do as much as you like, and we will truncate to 100 as the final step in computing the grade. The individual tasks, point value, and example objects are:

Roller Coasters Check Point:

Technique	Details	Points
Arc-Length Parameterization	Having your train move at a constant velocity (rather than moving at a constant change of parameter value) makes things better. Implementing this is an important step towards many other advanced features. You should allow arc-length parameterization to be switched on and off to emphasize the difference, you should also provide a speed control.	5
Approximating C2 curve	To draw the curves (or to compute arc length parameterizations), you need to sample along the curve (for example, to draw lines connecting the points). The curves are simple enough that simply sampling them uniformly and densely is practical.	5
Draw nicer looking tracks	The most basic track is just a line. To make something nicer (to make a tube or a ribbon), you need to consider the geometry of the curve. <ul style="list-style-type: none"> Parallel rails: for parallel rails, simply offsetting the control points (in world space) doesn't work. You need to know the local coordinates as you go around the track. Rail ties: ties are the cross pieces on railroad tracks. Getting them right (uniformly spaced) requires good arc-length parameterization. In the example code, you can turn the arc-length parameterization on and off to see the difference. 	5
Correct Orientation in 3D	The simple schemes for orienting the train break down in 3D - in particular, when there are loops. Make it so that your train consistently moves along the track (so its under the track at the top of a loop). One good way to provide for proper orientations is to allow the user to control which direction is "up" at points along the curve. This allows you to do things like corkscrew roller coasters. The sample solution does this (its why the framework has an orientation vector for each control point). Note that the train still needs to face forward, the given orientation is just a hint as to which way up should be.	5

Water Simulation Check Point:

Technique	Details	Points
Sine Wave	Implement the sine wave on the surface of water.	5
Height maps	Implement the height map method on the surface of water.	5
Skymapping reflection	Combination of refraction and reflection with an environment map for the sky and a set of texture map for the box.	5
Refraction map and reflection	Using the refraction map and reflection map to simulate the possible viewing condition from above the water to simulate the refraction and reflection effects.	5

If there's something that you want to do that you think is a good task, but not listed, please ask. We may extend this list at a later date as we think of more ideas. We may not be able to help you with some of these, so doing it will require some bravery and determination.

It is important that if you do something, you are able to show it off in the demo/make a picture for your album. So if you model some nice object, make sure there's a fast way to get the camera to go there. Or, if you do subdivision, show different levels of the same object so we can tell you really did the subdivision.

Some of the challenges require something to be complicated. You might wonder "when is an animation complicated enough that it qualifies as a challenge" or something like that. Generally, if you have any doubt, then it probably isn't so complicated. But if you are in doubt, ask.

Advanced Features

To get a better grade, and to really make the assignment fun, you should add some advanced features to your train or amusement part.

Note: the exact point values for each of these is not given. The rough guide here will give you some relative importances (big features are worth more than small ones).

We will only check the features that you say that you have implemented correctly. Partial credit will be given for advanced features, but negative credit may be given for really incorrect features. (so, its

better to not say you implemented a feature than to show us something that is totally wrong).

Also, remember that in your demo, you will have to show off the feature, so think about what demonstration will convince us that it works. For example, with arc-length parameterization, you're best off being able to switch it on and off (so we can compare with the normal parameterization), and think about a track that really shows off the differences. You should probably turn in example tracks that show off the features.

Technique	Requirement	Points	Suggestion
Tension control of a spline	<ul style="list-style-type: none"> Tension parameter controls how loosely or tightly the cardinal spline fits the input control points. 	2.5	Track
Multiple cars	<ul style="list-style-type: none"> Having multiple cars on your train (that stay connected) is tricky because you need to keep them the correct distance apart. You also need to make sure that the ends of the cars are on the tracks (even if the middles aren't) so the cars connect. 	2.5	Train
Real Train Wheels	<ul style="list-style-type: none"> Real trains have wheels at the front and back that are both on the track and that swivel relative to the train itself. If you make real train wheels, you'll need arc-length parameterization to keep the front and rear wheels the right distances apart (make sure to draw them so we can see them swiveling when the train goes around a tight turn). In the sample solution, the wheels are trucked (they turn independently), but each car still rotates around its center (so its as if they are floating above the wheels). You can do better than that. 	2.5	Train
Simple Physics	<ul style="list-style-type: none"> Roller coasters do not go at constant velocities - they speed up and slow down. Simulating this (in a simple way) is really easy once you have arc-length parameterization. Remember that Kinetic Energy - Potential Energy should remain constant (or decrease based on friction). This lets you compute what the velocity should be based on how high the roller coaster is. Even Better is to have "Roller Coaster Physics" - the roller coaster is pulled up the first hill at a constant velocity, and "dropped" where it goes around the track in "free fall." You could even have it stop at the platform to pick up people. Note: you should implement arc-length first. Once you get it right it is much easier. 	2.5	Train
Adaptive subdivision	<ul style="list-style-type: none"> To draw the curves (or to compute arc length parameterizations), you need to sample along the curve (for example, to draw lines connecting the points). The curves are simple enough that simply sampling them uniformly and densely is practical. Adaptive sampling (when the curve is straight, fewer line segments are needed) is a better approach, but the benefits may be hard to see. If you implement adaptive sampling, be sure to have some way to show off that it really works. 	2.5~5	Track
Make totally over-the-top tracks	<ul style="list-style-type: none"> This is more of a piece of artwork, but to do something really fancy, you'll probably write code to create the points. If you're really into trains, you could have different kinds of cars. In particular, you could have an engine and a caboose. Switches and more complex connections in the layout 	5	Track
Multiple tracks and trains	<ul style="list-style-type: none"> The track could have branches, ... You'd need to have some way to tell the trains which way to go, and some way to deal with branching curves. The framework is 	5	Track, Train

	pretty much set up to have one track and train, but you could change this without too much hassle. You would need to make multiple World objects, but the hard part would be adapting the UI.		
Sketch-based interface	<ul style="list-style-type: none"> Allow the user to sketch a rough shape, and then create a smooth curve from that. This is difficult to do well, but if you're interested, we can suggest some interesting things to try. 	10	Track
Have People on your Roller Coaster	<ul style="list-style-type: none"> Little people who put their hands up as they accelerate down the hill are a cool addition. (I don't know why putting your hands up makes roller coasters more fun, but it does). The hands going up when the train goes down hill is a requirement. 	2.5	Train
Headline	<ul style="list-style-type: none"> Have the train have a headlight that actually lights up the objects in front of it. This is actually very tricky since it requires local lighting, which isn't directly supported. 	2.5	Train
Particle system	Model a complex, moving object as a set of little particles. You can make fireworks, rain, snow, fountains, fire, ...	5~12.5 (5 for first, 2.5 for each extra one)	Foundation, firework, ...
Non-flat terrain	<ul style="list-style-type: none"> This is mainly interesting if you have the train track follow the ground (maybe with tressles or bridges if the ground is too bumpy). 	5	Terrain
Support Structure	<ul style="list-style-type: none"> When the track is in the air, you could create tressles or supports to hold it up (like a real roller coaster). Of course, you'd want to handle the case where the track crosses. 	2.5	Track
Scenery	<ul style="list-style-type: none"> Having other (non-moving) objects in the world gives you something to look at when you ride the train. 	2.5	Scenery
Tunnels	<ul style="list-style-type: none"> Make hills with tunnels through them for your train to go through. The tunnel should adapt its shape to the track (so it should curve like the track curves). 	2.5	Scenery
Texture Mapping	<ul style="list-style-type: none"> You must create your own texture (at least 3) to earn the point from this score. Add texture mapped polygonal objects to the environment. Each "object" for grading purposes consists of at least 5 polygons all texture mapped. Different objects require different maps. 	2.5~5	Buildings, walls, roadways Hierarchical Animated Model
Parametric Instancing	<ul style="list-style-type: none"> Add an object described by parameters. You must create multiple instances with different parameters, and each class of model counts for separate points, not each instance. 	5	Trees (cones on sticks), buildings, even rides
Sweep Objects(other than rail)	<ul style="list-style-type: none"> Add an object created as a sweep, either an extrusion or a surface of revolution. The important thing is that it be created by moving some basic shape along a path. The overall object must use at least three different uses of the swept polygon. In other words, something like a cylinder isn't enough, but something like two cylinders joined to form an elbow is. 	2.5	trash bins, trees
Something cools	Yes, you might think of something to do that we didn't mention here. If its really cool, we might give you points for it. We'd like you to focus on trying to do more with the curves aspect of this assignment (rather than making arbitrary eye-candy), so we won't give you points for just making eye candy (e.g. putting textures on things) - there will be a whole project devoted to that. If you want to do something and you want to make sure it will be worth points, send the instructor email. In the past people have come up	2.5 ~ 10	...

	with crazy stuff - some of them have become part of the assignment.		
Really Cool Shaders	<ul style="list-style-type: none"> Graphics Processing Unit(GPU) has become very important aspect in graphics. We would like to explore the usage of them such as environment map, particle simulation, water simulation, and so on. Everyone has to write 3 shaders. But if the shaders do something really cool, that counts for technical challenge. Bump Mapping definitely counts as a technical challenge. A properly anti-aliased procedural shader would be a technical challenge. Phong shading would be a technical challenge except that you can get the code from just about anywhere - combine it with something more imaginative to make a technical challenge. We'll give technical challenge points for really imaginative shaders. 	2.5 ~ 7.5	Foundation, firework, ...
Hack Rendering Tricks	<ul style="list-style-type: none"> Local lights (2.5~5): have a light that only effects nearby objects. You can't just do this using the OpenGL falloff since that limits the number of lights you have - you'll need to switch lights on and off depending on what object is being drawn. Note: local lighting is NOT the hack "spotlight cones" that my sample program does. Consider putting a flashing (or even spinning) siren on a police car, or ... Inter-object shadows and reflections (2.5-7.5): Shadows on the groundplane are easy. Shadows cast from one object onto another are much harder. Reflections of actual (dynamic) objects are really tricky - as opposed to using environment mapping with static environments. Implementing shadow mapping (or shadow volumes) is one way to do this. 	2.5 ~ 12.5 (Depends on TA's decision)	...
Non-Photorealistic Rendering	Give your world an artistically styled look to the drawing. For example, make everything look like a pencil drawing by tracing object edges and making things squiggly, or use "toon shading" to make things look like a cartoon. Note: if you are really going to do an NPR world, we might be willing to remove the texture requirements - but only if you'll be doing enough NPR stuff.	5 ~ 10(Depends on TA's decision)	Foundation, firework, ...
Very Advanced Texturing	<ul style="list-style-type: none"> Skybox (2.5): make a textured sky - have clouds and stars (at night). But note that a proper skybox stays fixed relative to the camera (so it doesn't have to be so huge that it causes Z-Buffer issues). Billboard Object (2.5): model a complex shape by using a flat object that moves to face the viewer (and probably transparency). Nice trees can be done this way. Projector Textures (2.5): make a slide projector or something that creates an unusual effect. To get full credit, it needs to be clear that you are using the texture matrix stack to get a projection. Environment Map (2.5~5): Use environment mapping to create a reflective surface. Be sure to describe how you made the map. 	2.5 ~ 12.5	...
Artistic Points	About how beautiful or cool is your amusement. It is a good decision to decorate your amusement part with a theme.	2.5 ~ 12.5	...

Submission

Upload it to the FTP site.

What to turn in

By the deadline you must turn in:

- Everything needed to compile your program (.cpp files, .H files, .vcproj files, .sln files, and UI files or other things your program needs). Be sure to test that your program can be copied out of this directory and compiled on a Storm computer.
- if you work with a partner, only turn in one copy of the project. In the other person's directory put a single file in your handing directory - a README.txt that says where to look.
- Your README file.
- You should make a subdirectory of the project directory called "Gallery." In this directory, please put a few JPG pictures of the best scenes in your town. Please name the pictures login-X.jpg (where X is a number). Put a text file in the directory with captions for the pictures. (note: to make pictures, use the screen print and then use some program to convert them to JPG).
- You must also make a subdirectory of the project directory called "Video" to put a 1~2 minutes of video capturing your world.
- Some example track files. You should not turn in the track files that we distribute (we give you a bunch) - only turn in ones that you made.

Documentation

In your readme, please make sure to have the following (you can break it into separate files if you prefer):

1. A abstract of your work
2. Instructions on how to use your program (in case we want to use it when you're not around)
3. A list of all the features that you have added, including a description, and an explanation of how you know that it works correctly.
4. An explanation of the types of curves you have created
5. A discussion of any important, technical details (like how you compute the coordinate system for the train, or what method you use to compute the arc length)
6. A list of the objects you modeled (if you made lots of different objects, just list the 5-10 most interesting ones). Please order the list so the most complicated/impressive one is first.
7. A list of the behaviors you made. Please order the list so the most complicated/impressive one is first.
8. A list of the shaders that you made with a brief description of each. A list of the technical challenges that you attempted / completed, with a description of what you did and what you used it for.
9. Any non-standard changes that you make to the code
10. If you used the sample, code, a file that describes any changes you made to the "core" of the system (e.g. other than changing main.cpp and adding new Objects and Behaviors).
11. If you did not use the example code, an explanation of why you chose not to, and a discussion of your program's features.
12. Anything else we should know to compile and use your program

Some Hints

Use the framework. It will save you lots of time.

In case it isn't obvious, you will probably use Cardinal Cubic splines (like Catmull-Rom splines). Cubic Bezier's are an option (just be sure to give an interface that keeps things C1. For the C2 curves, Cubic B-Splines are probably your best bet.

You should make a train that can move along the track. The train needs to point in the correct direction. It is acceptable if the center of the train is on the track and pointing in the direction of the tangent to the track. Technically, the front and back wheels of the train should be on the track (and they swivel with respect to the train). If you implement this level of detail, please say so in your documentation. It will look cool.

In order to correctly orient the train, you must define a coordinate system whose orientation moves along with the curve. The tangent to the curve only provides one direction. You must somehow come up with the other two directions to provide an entire coordinate frame. For a flat track, this isn't too difficult. (you know which way is up). However, when you have a roller coaster, things become more complicated. In fact, the sample solution is wrong in that it will break if the train does a loop.

The sample solution defines the coordinate frame as follows: (note: you might want to play with it understand the effects)

1. The tangent vector is used to define the forward (positive Z) direction.
2. The "right" axis (positive X) is defined by the cross product of the world up vector (Y axis) and the forward vector.
3. The local "up" axis is defined by the cross product of the first two.

Doing arc-length parameterizations analytically is difficult for cubics. A better approach is to do them numerically. You approximate the curve as a series of line segments (that we know how to compute the length of). A simple way to do it: create a table that maps parameter values to arc-lengths. Then, to compute a parameter value given an arc length, you can look up in the table and interpolate.

Alternatively, you can do a little search to compute the arc length parameterization. If you have a starting point (u) in parameter space, and a distance you want to move forward in arc length space, you can move along in parameter space, compute the next point, find the distance to it, and accumulate.

Suggestions

- Have fun and be inventive.
- A key thing to consider is polygon count. Graphics hardware can only display so many polygons in a second, and if you try to display too many the frame rate will collapse. Texture maps also use memory, so too many textures can even more dramatically affect performance.
- The way the train alignment is set up, it is simplest to do just a single carriage, and a short one at that. Doing lots of cars makes it harder to keep them on the track, although it is possible.
- Make use of the OpenGL error checking mechanism. It is described in the OpenGL Programming Guide.
- Start simple - just try to get a polygon to appear in the center of the world.
- The way the current carriage transformations are set up, the origin for the train is assumed to be at the bottom (at track level).
- It is OK to have multiple modeling techniques in one object. For instance, you could have a carriage made up of some texture mapped polygons with some subdivision areas. You get all the points if you do a sufficient amount of each technique.
- It is OK to borrow code from other sources - but not other students. You will probably learn as much trying to figure out how someone else's code works as you would doing it yourself.
- Texture images abound on the web, so feel free to use them. Or you can use a program like Photoshop to create your own. You might even find a use for the first project.

Results



林育生, and 郭俊廷, A Tiny Amusement Park with Roller Coasters and Water Shaders -- 林育生_郭俊廷

[Detail](#)



陳泳峰, and 陳宥潤, A Tiny Amusement Park with Roller Coasters and Water Shaders -- 陳泳峰_陳宥潤

[Detail](#)



留希哲, A Tiny Amusement Park with Roller Coasters and Water Shaders -- 留希哲

[Detail](#)



韓悅華, and 陳洛翔, A Tiny Amusement Park with Roller Coasters and Water Shaders -- 韓悅華_陳洛翔

[Detail](#)

3D 遊戲設計教材節錄

This Note

- Graphics Toolkit
- Coordinate system
- 3D Transformations
- Directions
- Rotation
- Details of Transformation is in Prof. Yao's Fundamental of CG
- Project 1 Due & Demo

Computer Graphics
NTUST CSIE Laboratory

Question

- How do the graphics pipeline compute it?

Computer Graphics
NTUST CSIE Laboratory

The Graphics Process

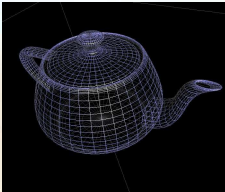

```

    graph TD
      A[3D Geometric Models] --> E[Rendering]
      B[3D Animation Definition] --> E
      C[Lighting Information] --> E
      D[Texture Information] --> E
      E --> F[Image Storage & Display]
    
```

Computer Graphics
NTUST CSIE Laboratory

Rendering Primitives

- Use **graphics hardware** for real time...
- These can render points, lines, and triangles.
- A surface is thus an **approximation** by a number of such primitives.

Computer Graphics
NTUST CSIE Laboratory

Traditional Rendering Pipeline

Computer Graphics
NTUST CSIE Laboratory

Traditional Rendering Pipeline

- What's wrong with this model (for an OpenGL system)?
- Model/view transforms combined
- Really "vertices" not "primitives"
 - Making this the *vertex pipeline*
- There's a lot going on in the "scan conversion" stage!
 - Primitive assembly
 - Rasterization
 - Texture mapping
 - Per-pixel lighting
 - Visibility (Z-buffer)
- We refer to these collectively as the *pixel or fragment pipeline*

Computer Graphics
NTUST CSIE Laboratory

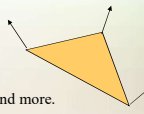
Rendering

- Generate an image **showing** the contents of some region of space
 - The region is called the **view volume**, and it is defined by the user
- Determine **where each object** should go in the image
 - *Viewing, Projection*
- Determine **which pixels** should be filled
 - *Rasterization*
- Determine **which object** is in front at each pixel
 - *Hidden surface elimination, Hidden surface removal, Visibility*
- Determine **what color** it is
 - *Lighting, Shading*

Computer Graphics
NTUST CSIE Laboratory

What's a "3D scene"?

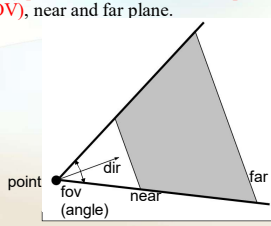
- First, of all to take a picture, it takes a camera – a **virtual one**.
 - **Decides what** should end up in the final image
- A 3D scene is:
 - **Geometry** (triangles, lines, points, and more **primitives**)
 - **Light sources**
 - **Material** properties of geometry
 - **Textures** (images to glue onto the geometry)
 - ...
- A triangle consists of 3 vertices
 - A vertex is 3D position, and may include normals and more.



Computer Graphics
NTUST CSIE Laboratory

Virtual Camera

- Defined by **position**, **direction vector**, **up vector**, **field of view(FOV)**, near and far plane.

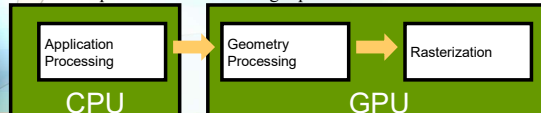
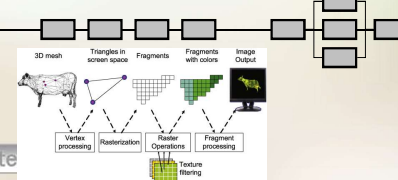


- Create image of geometry **inside gray region**
- Used by OpenGL, DirectX, ray tracing, etc.

Computer Graphics
NTUST CSIE Laboratory

High-Level Pipeline

- Back up & think about the larger picture:

Computer Graphics
NTUST CSIE Laboratory

The APPLICATION Stage

- Executed on the CPU
 - Means that the programmer **decides what happens here**
- Examples:
 - Collision detection
 - Speed-up techniques
 - Animation
- Most important task: send rendering primitives (e.g. triangles) to the graphics hardware

Computer Graphics
NTUST CSIE Laboratory

Application Geometry Rasterizer

The GEOMETRY Stage

- Task: "**geometrical**" operations on the input data (e.g. triangles)
- Allows:
 - **Move objects** (matrix multiplication)
 - **Move the camera** (matrix multiplication)
 - Compute **lighting** at vertices of triangle
 - **Project onto screen** (3D to 2D)
 - **Clipping** (avoid triangles outside screen)
 - **Map to window**

Computer Graphics
NTUST CSIE Laboratory

Application Geometry Rasterizer

Getting Geometry on the Screen

- Given geometry positioned in the **world coordinate system**, how do we get it **onto the display**?
 - Transform to camera coordinate system
 - Transform (warp) into **canonical view volume**
 - Clip
 - Project to display coordinates**
 - Rasterize

Computer Graphics
NTUST CSIE Laboratory

Pipeline of Transformations

Standard sequence of transforms

Computer Graphics
NTUST CSIE Laboratory

Animate Objects and Camera

- Can animate in many different ways with **4x4 matrices** (topic of next lecture)
- Example:
 - Before **displaying a torus** on screen, a matrix that represents a rotation can be applied. The result is that the torus is rotated.
 - Same thing with camera (this is possible since motion is relative)

Computer Graphics
Application Geometry Rasterizer
NTUST CSIE Laboratory Tomas Akenine-Möller

3D Viewing Process

Transform into viewport in 2D device coordinates for display

Computer Graphics
NTUST CSIE Laboratory

The RASTERIZER Stage

- Main task: take output from GEOMETRY and turn into visible pixels on screen

- Also, add **textures** and various other per-pixel operations
- And **visibility** is resolved here: sorts the primitives in the z-direction

Computer Graphics
Application Geometry Rasterizer
NTUST CSIE Laboratory

High-Level Pipeline

Application	Geometry (a.k.a. "vertex pipeline")	Rasterization (a.k.a. "pixel pipeline" or "fragment pipeline")
Handle input	Transform	Rasterize (fill pixels)
Simulation & AI	Lighting	Interpolate vertex parameters Look up/filter textures
Culling	Skinning	Z- and stencil tests
LOD selection	Calculate texture coords	Blending
Prefetching		

Computer Graphics
NTUST CSIE Laboratory

Rewind! Let's take a closer look

- The programmer "sends" down primitives to be rendered through the pipeline (using API calls)
- The geometry stage does **per-vertex** operations
- The rasterizer stage does **per-pixel** operations
- Next, **scrutinize** geometry and rasterizer

Computer Graphics NTUST CSIE Laboratory

The GEOMETRY stage in more detail

- The model transform**
- Originally, an object is in "model space"
- Move, orient, and transform geometrical objects into "world space"
- Example, a sphere is defined with origin at (0,0,0) with radius 1
 - Translate, rotate, scale to make it appear elsewhere
- Done per vertex with a 4x4 matrix multiplication!
- The user can apply different matrices over time to animate objects

Computer Graphics NTUST CSIE Laboratory

GEOMETRY: The View Transform

- You can move the camera in the same manner
- But apply inverse transform to objects, so that camera looks down negative z-axis

The diagram illustrates the view transform. On the left, a camera (represented by a circle with a lens) is positioned to look at a 3D scene containing a triangle, a star, and a square. An arrow points to the right, where the same scene is shown from the camera's perspective, but the objects are now oriented as if they were looking down the negative z-axis of a coordinate system (x, y, z).

Computer Graphics NTUST CSIE Laboratory

GEOMETRY: Lighting

- Compute "lighting" at vertices

The diagram shows the lighting process. A light source (a dot labeled 'light') is positioned above a triangle. The triangle's vertices are labeled 'blue', 'red', and 'green'. An arrow labeled 'Geometry' points from the triangle to a box containing the same triangle with the vertex colors. Another arrow labeled 'Rasterizer' points from this box to a final image showing a smooth color gradient across the triangle.

- Try to mimic how light in nature behaves
 - Hard to use empirical models, hacks, and some real theory
- Much more about this in later lecture

Computer Graphics NTUST CSIE Laboratory

The Full Story

- We have only touched on the **complexities of illuminating surfaces**
 - The common model is hopelessly inadequate for accurate lighting (but it's fast and simple)
- Consider two sub-problems of illumination
 - Where does the light go? *Light transport*
 - What happens at surfaces? *Reflectance models*
- Other algorithms address the transport or the reflectance problem, or both
 - Much later in class, or a separate course Computer Rendering

Computer Graphics NTUST CSIE Laboratory

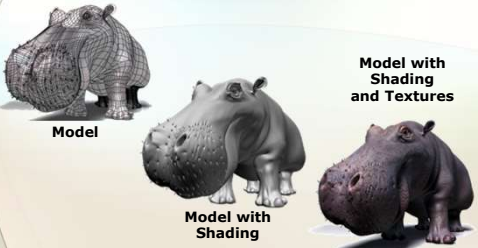
Compute lighting at vertices, then interpolate over triangle

The diagram shows the lighting process. A light source (a dot labeled 'light') is positioned above a triangle. The triangle's vertices are labeled 'blue', 'red', and 'green'. An arrow labeled 'Geometry' points from the triangle to a box containing the same triangle with the vertex colors. Another arrow labeled 'Rasterizer' points from this box to a final image showing a smooth color gradient across the triangle.

- How compute lighting?
- We could set colors per vertex manually
- For a **little** more realism, compute lighting from
 - Light sources
 - Material properties
 - Geometrical relationships

Computer Graphics NTUST CSIE Laboratory

The Quest for Visual Realism



Model

Model with Shading

Model with Shading and Textures

Copyright©1997, Jeremy Birn

Computer Graphics
NTUST CSIE Laboratory


The Limits of Geometric Modeling

- Although graphics cards can render **over 10 million polygons per second**, that number is **insufficient** for many phenomena
 - Clouds
 - Grass
 - Terrain
 - Skin

Computer Graphics
NTUST CSIE Laboratory

Beautification of Surfaces

- Texture mapping (ubiquitous in hardware)
 - Paste **photograph or bitmap** on a surface to provide detail (e.g. brick pattern, sky with clouds, etc.)
 - Think of a texture map as **contact paper**, but made of stretchable latex
 - Map **texture/pattern pixel array onto surface** to replace (or modify) original color; can still use original intensity to modulate texture
 - The function is called **texture map** and the process is called **texture mapping**.




Sphere with no texture

Texture image

Sphere with texture

Wikipedia




Microsoft Flight Simulator

Computer Graphics
NTUST CSIE Laboratory

Motivation to Use Texture Mapping (1/2)

How do we increase the amount of detail?

- Expensive solution:** add more detail to model
 - detail **incorporated** as a part of object
 - modeling tools **aren't very good** for adding detail
 - model takes **longer to render**
 - model takes up **more space** in memory
 - complex detail **cannot be reused**
- Efficient solution:** map a texture onto model
 - texture maps can be **reused**
 - texture maps take up space in memory, but can be **shared**, and **compression** and **caching** techniques can reduce overhead significantly compared to real detail
 - texture mapping can be **done quickly** (we'll see how)
 - placement and creation of texture maps can be made **intuitive** (e.g., tools for adjusting mapping, painting directly onto object)
 - texture maps do not **affect the geometry** of the object



Computer Graphics
NTUST CSIE Laboratory

Motivation to Use Texture Mapping (2/2)

- What **kind of detail** goes into these maps?
 - Diffuse, ambient and specular **colors**
 - Specular exponents
 - Transparency, reflectivity
 - Fine detail** surface normals (bumps)
 - Data to visualize
 - Projected lighting and shadows
 - Games use "billboards" for distant detail. (sprites are effectively moving billboards)

Computer Graphics
NTUST CSIE Laboratory

Texture Maps

- How is texture **mapped to the surface**?
 - Dimensionality: 1D, 2D (image), 3D (solid)
 - Procedural** v.s. table **look-up**
 - Texture coordinates (**s,t**)
 - Surface parameters (**u,v**)
 - Projection: spherical, cylindrical, planar
 - Reparameterization** (Prof. Yao's Class)
- What does texture **control**?
 - Surface color and transparency
 - Illumination: environment maps, shadow maps
 - Reflection function: reflectance maps
 - Geometry: displacement and bump maps

Computer Graphics
NTUST CSIE Laboratory

Texture Maps

Tom Porter's Bowling Pin

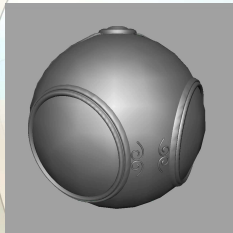
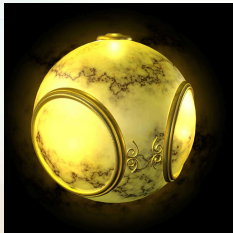






Computer Graphics
NTUST CSIE Laboratory



Texture Mapping

geometric model texture mapped

Computer Graphics
NTUST CSIE Laboratory




Texture Mapping

2D mapping 3D mapping

Computer Graphics
NTUST CSIE Laboratory

Decal Textures

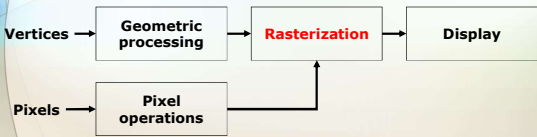




Copyright © 2003, Technion-Israel Institute of Technology

Computer Graphics
NTUST CSIE Laboratory

Where Does Mapping Take Place?

- Mapping techniques are implemented **at the end of the rendering pipeline**
- Very efficient because few polygons pass down the geometric pipeline



Computer Graphics
NTUST CSIE Laboratory

Texture Pipeline

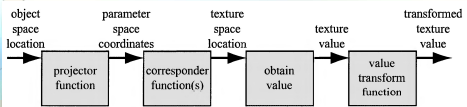


Figure 6.2. The generalized texture pipeline for a single texture.

- Object space location:** the location in object space
- Parameter space coordinate:** the coordinate in texture space
- Texture space location:** clamp to (0, 1) position
- Texture value:** the 4-column data store in the texture map
- Transformed texture value:** RGB, RBG_a, Normal, ...

Computer Graphics
NTUST CSIE Laboratory

GEOMETRY: Projection

- Two major ways to do it
 - Orthogonal (useful in few applications)
 - Perspective (most often used)
 - Mimics how humans perceive the world, i.e., objects' apparent size decreases with distance

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

GEOMETRY: Projection

- Also done with a matrix multiplication!
- Pinhole camera (left), analog used in CG (right)

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

GEOMETRY: Clipping and Screen Mapping

- Square (cube) after projection
- Clip primitives to square

- Screen mapping, scales and translates square so that it ends up in a rendering window
- These "screen space coordinates" together with Z (depth) are sent to the rasterizer stage

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

GEOMETRY: Summary

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER in More Detail

- Scan-conversion
 - Find out which pixels are inside the primitive
- Texturing
 - Put images on triangles
- Interpolation over triangle
- Z-buffering
 - Make sure that what is visible from the camera really is displayed
- Double buffering
- And more...

Tessellation Pipeline

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Scan Conversion

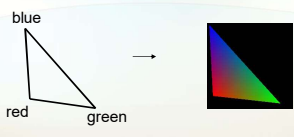
- Triangle vertices from GEOMETRY is input
- Find pixels inside the triangle
 - Or on a line, or on a point
- Do per-pixel operations on these pixels:
 - Interpolation
 - Texturing
 - Z-buffering
 - And more...

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Interpolation

- Interpolate colors over the triangle
 - Called **Gouraud** interpolation




Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Texturing

- One application of texturing is to "glue" images onto geometrical object



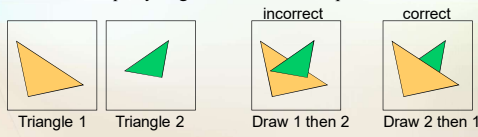
- Uses and other applications
 - More realism
 - Bump mapping
 - Pseudo reflections
 - Store lighting
 - Almost infinitely many uses

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Z-buffering

- The graphics hardware is pretty stupid
 - It "just" draws triangles
- However, a triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



Triangle 1 Triangle 2 Draw 1 then 2 Draw 2 then 1

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Z-buffering

- Would be nice to avoid sorting...
- The Z-buffer (aka depth buffer) solves this
- Idea:
 - Store z (depth) at each pixel
 - When scan-converting a triangle, compute z at each pixel on triangle
 - Compare triangle's z to Z-buffer z-value
 - If triangle's z is smaller, then replace Z-buffer and color buffer
 - Else do nothing
- Can render in any order

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER—Double Buffering

- The monitor displays one image at a time
- So if we render the next image to screen, then rendered primitives pop up
- And even worse, we often clear the screen before generating a new image
- A better solution is "double buffering"

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

RASTERIZER: Double Buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

Computer Graphics NTUST CSIE Laboratory

Application Geometry Rasterizer

And, lately...

- Don't forget! Not your father's GPU
- Programmable shading has **become a hot topic**
 - Vertex shaders
 - Pixel shaders
- Adds more control and much more possibilities for the programmer

Computer Graphics
NTUST CSIE Laboratory

Programmable Pipelines

- The amount of programmability is increasing by leaps and bounds
 - Allow you to **write a small program** that determines how the color of a vertex or pixel is computed
 - Your program has access to the surface normal and position, plus anything else you care to give it (like the light)
 - You can add, subtract, take dot products, and so on
 - **Vertex shaders:** more instructions, variable looping, branching, subroutines
 - **Pixel shaders:** still SIMD, but with more instructions, unlimited texture accesses, pixel kill (good for lighting computation)
- The data formats are also improving
 - IEEE floating point throughout the pixel pipeline!
 - Various versions

Computer Graphics
NTUST CSIE Laboratory David Luebke

Graphics Pipeline: GPU

Note:

- Vertex processor does all transform and lighting
- Pipe widths vary
- Intra-GPU pipes wider than CPU→GPU pipe
- Thin GPU→CPU pipe
- Many caches and FIFOs not shown

Here's what's cool:

- Can now program vertex processor!
- Can now program pixel processor!

Computer Graphics
NTUST CSIE Laboratory

First Innovation

- Current hardware allows you to break from the standard illumination model
- Programmable **Vertex shaders** and **Fragment shaders** allow you to **write a small program** that determines how the color of a vertex or pixel is computed
 - Your program has access to the surface normal and position, plus anything else you care to give it (like the light)
 - You can add, subtract, take dot products, and so on
- **Fragment shaders** are most useful for lighting because they operate on every pixel.

Computer Graphics
NTUST CSIE Laboratory

Original And Modified Pipeline

- Replace transform and lighting with **vertex shader**
 - Vertex shader must now do transform and lighting
 - But can also do more
- Replace texture stages with **fragment (pixel) shader**
 - Previously, texture stages were only per-pixel operations
 - Fragment shader must do texturing

Computer Graphics
NTUST CSIE Laboratory

Second Innovation

- **Geometry shaders** are introduced in Direct3D 10 and OpenGL 3.2;
 - They can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline.

Computer Graphics
NTUST CSIE Laboratory

Third Innovation: OpenGL 4.0

- Tessellation shaders are introduced in OpenGL 4.0 and Direct3D 11.
 - It adds two new shader stages to the traditional model
 - Tessellation Control Shaders (also known as Hull Shaders)
 - Tessellation Evaluation Shaders (also known as Domain Shaders)
 - For simpler meshes to be subdivided into finer meshes at run-time according to a mathematical function.
 - Related variables:
 - the distance from the viewing camera to allow active level-of-detail scaling.

Computer Graphics
NTUST CSIE Laboratory

Graphics Pipeline: GPU

Computer Graphics
NTUST CSIE Laboratory

Graphics Hardware / Interactive Rendering

- Key idea: **set of basic abstractions**
 - Z-buffer, texture, triangles, ...
- Implement these really well
- Let **programmers** figure out how to use it to do other things
- Expand **abstractions** based on what people figure out to do

Computer Graphics
NTUST CSIE Laboratory

OpenGL Rendering Pipeline

- Pipeline:** consists of **multiple stages**. Data flows in, being processed in each stages, then flows out
- Stages:** each stage **represents a unique function** to process the input data
 - Fixed function stages:** limited customization capability, typically exposes states for configuration
 - Programmable shader stages:** allow custom shader programs to be executed within, providing broader capability of customization

Computer Graphics
NTUST CSIE Laboratory

Fixed Function Pipeline (Legacy)

- Before OpenGL 3.0, OpenGL rendering is done in a fixed function pipeline
- Fixed pipeline is like a machine with a lot of switches/values to configure
- One cannot **change how the function** is implemented as well as the **order of execution**

Computer Graphics
NTUST CSIE Laboratory

Fixed Function Pipeline (Legacy)

Computer Graphics
NTUST CSIE Laboratory

Fixed Function Pipeline: Metaphor

OpenGL Fixed Function Pipeline

How do I press these buttons to get desired effect?

Computer Graphics
NTUST CSIE Laboratory

Programmable Pipeline

- Shader programs are introduced in OpenGL 2.0, and included in the core profile in OpenGL 3.0
- Fixed function pipeline is deprecated since OpenGL 3.0
- Shader programs, written in OpenGL Shading Language (GLSL), allow the programmers to customize certain stages in the OpenGL rendering pipeline

Computer Graphics
NTUST CSIE Laboratory

First-Modified Pipeline

- Replace transform and lighting with **vertex shader**
 - Vertex shader must now do transform and lighting
 - But can also do more
- Replace texture stages with **fragment (pixel) shader**
 - Previously, texture stages were only per-pixel operations
 - Fragment shader must do texturing

Computer Graphics
NTUST CSIE Laboratory

The First Generation

- Current hardware allows you to **break from** the standard illumination model
- Programmable *Vertex Shaders* and *Fragment Shaders* allow you to **write a small program** that determines how the color of a vertex or pixel is computed
 - Your program **has access to** the surface normal and position, plus anything else you care to give it (like the light)
 - You can add, subtract, take dot products, and so on
- **Fragment shaders** are most **useful for lighting** because they operate on every pixel

Computer Graphics
NTUST CSIE Laboratory

Vertex Shaders V.S. Pixel Shaders

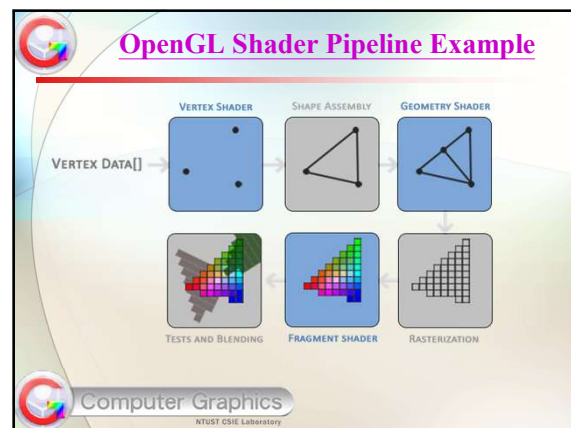
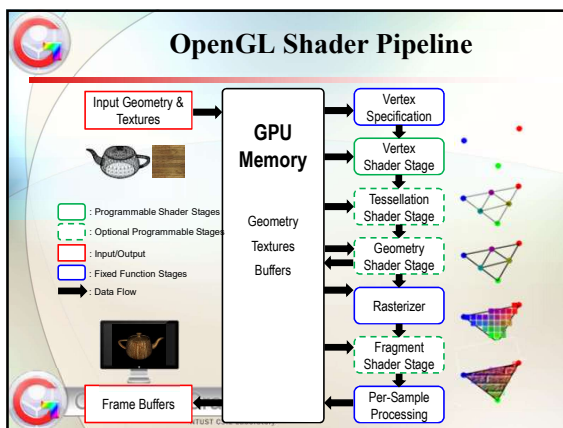
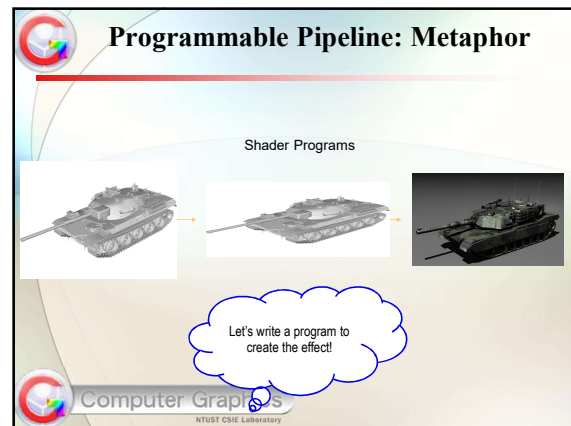
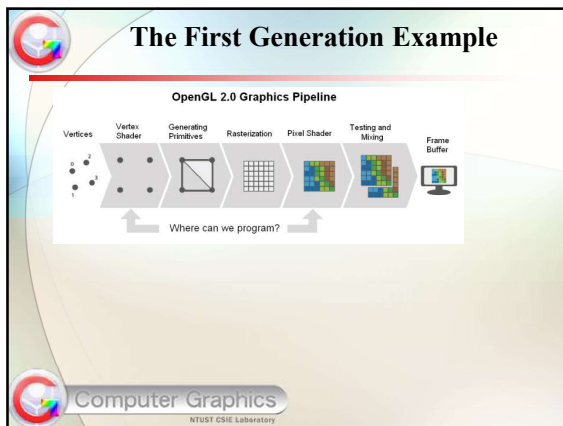
- Classical **Fixed-Function Pipeline (FFP)**: Per-Vertex Lighting, MVT + VT
 - Largely superseded on desktop by programmable pipeline
 - Still used in mobile computing
- Modern **Programmable Pipeline**: Per-Pixel Lighting
- **Vertex Shaders** (FFP and Programmable)
 - **Input**: per-vertex attributes (e.g., object space position, normal)
 - **Output**: lighting model terms (e.g., diffuse, specular, etc.)
- **Pixel Shaders** (Programmable Only)
 - **Input**: output of vertex shaders (lighting aka illumination)
 - **Output**: pixel color, transparency (R, G, B, A), and other stored in framebuffer.

Computer Graphics
NTUST CSIE Laboratory

Vertex Shaders V.S. Pixel Shaders

- Brief Digression
 - Note: vertices are *lit*, pixels are *shaded*
 - “Pixel shader”: well-defined (iff “pixel” is)
 - “Vertex shader”: misnomer (somewhat)
 - Most people refer to both as “shaders”

Computer Graphics
NTUST CSIE Laboratory



Examples: Shadow Mapping

- Ways to Handle Shadows
 - Projected planar shadows: works well on flat surfaces only
 - Shadow stencil buffer: powerful, excellent results possible; hard!

Projected planar shadows, Shadow volumes, Light maps, Hybrid approaches, Shadow Stencil Buffer

- OpenGL Shadow Mapping Tutorials
 - Beginner/Intermediate (Baker, 2003): <http://bit.ly/e1LA2N>
 - Advanced (Octavian et al., 2000): <http://bit.ly/flrYB> (old NeHe #27)

Computer Graphics
NTUST CSIE Laboratory


Examples: Reflection / Environment Mapping

- How To Create Direction Maps
 - Latitude-Longitude (Map Projections) - paint
 - Gazing Ball - photograph reflective sphere
 - Fisheye Lens - standard (wide-angle) camera lens
 - Cubical Environment Map - rendering program or photography
 - Easy to produce
 - "Uniform" resolution
 - Simple texture coordinates calculation

Computer Graphics
NTUST CSIE Laboratory

Examples: Reflection / Environment Mapping

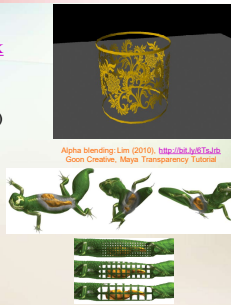
- Old NeHe OpenGL Mapping Tutorials (2000)
 - #6 (texture map onto cube) – Beginner (Molofee): <http://bit.ly/gKj2Nb>
 - #23 (sphere) – Intermediate (Schmick & Molofee): <http://bit.ly/e3Zb8h>
- nVidia Tutorial: OpenGL Sphere Map (1999): <http://bit.ly/eJEdAM>
- Issues: Non-Linear Mapping, Area Distortion, Converting Between Maps



Computer Graphics NTUST CSIE Laboratory

Examples: Transparency Map

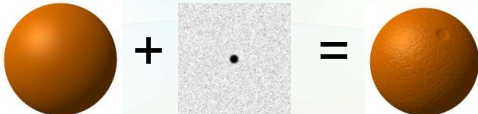
- OpenGL Transparency How-To at OpenGL.org: <http://bit.ly/hRaQgk>
- Screen Door Transparency
 - Use `glPolygonStipple()`, `glEnable(GL_POLYGON_STIPPLE)`
 - See <http://bit.ly/glhOpJ>
- Glass-Like Transparency using Alpha Blending
 - Use `glEnable(GL_BLEND)`, `glBlendFunc(...)`
 - See <http://bit.ly/hs82Za>



Computer Graphics NTUST CSIE Laboratory

Examples: Bump Mapping

- Goal: Create Illusion of Textured Surface

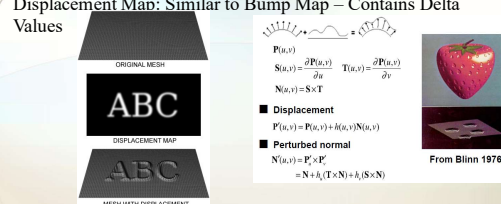


- Idea
 - Start with **regular smooth object**
 - Make **height map** (by hand and/or using program, i.e., procedurally)
 - Use map to **perturb surface normals**
 - Plug new normals into illumination equation
- Tutorial for OpenGL (Baker, 2003): <http://bit.ly/fun4a5>

Computer Graphics NTUST CSIE Laboratory

Examples: Displacement Mapping

- Displacement Map: Similar to Bump Map – Contains Delta Values



$$P(u,v) = P(u,v) + h(u,v)N(u,v)$$

$$N(u,v) = N \times T = N_x T_x + N_y T_y + N_z T_z$$

Displacement Mapping © 2005 Wikipedia
http://en.wikipedia.org/wiki/Displacement_mapping
 Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
<http://bit.ly/h2h2uZ> (CS 348B)

- Displacement Mapping: Uses Open GL Shading Language (GLSL)
- Tutorial using GLSL (Guinot, 2006): <http://bit.ly/dWXNya>

Computer Graphics NTUST CSIE Laboratory

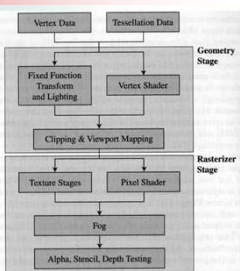
OpenGL Shading (Overview)

- Set Up Point Light Sources
 - Directional light given by 'position' vector


```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
glLight(GL_LIGHT0, GL_POSITION, light_position);
```
 - Point source given by 'position' point


```
GLfloat light_position[] = {1.0, 1.0, 1.0, 1.0};
glLight(GL_LIGHT0, GL_POSITION, light_position);
```
- Set Up Materials, Turn Lights On

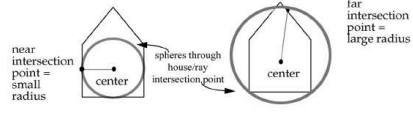

```
GLfloat mat_specular[] = {0.0, 0.0, 0.0, 1.0};
GLfloat mat_diffuse[] = {0.8, 0.6, 0.4, 1.0};
GLfloat mat_ambient[] = {0.8, 0.6, 0.4, 1.0};
GLfloat mat_shininess[] = {20.0};
glMaterial(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterial(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterial(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterial(GL_FRONT, GL_SHININESS, mat_shininess);
glShadeModel(GL_SMOOTH); //enable smooth shading ?
glEnable(GL_LIGHTING); // enable lighting ?
glEnable(GL_LIGHT0); // enable light 0 ?
```
- Start Drawing (`glBegin ... glEnd`)



Computer Graphics NTUST CSIE Laboratory

Texturing – Object Center Method

When we treat the object intersection point as a point on a sphere, our "sphere" won't always have the same radius



- What radius to use?
 - Compute the radius as the distance from the center of the complex object to the intersection point. Use that as the radius for the (u, v) mapping.

Computer Graphics NTUST CSIE Laboratory

OpenGL Texturing

In initialization:

```
glGenTextures(...);
glBindTexture( ... );
glTexParameter(...); glTexParameter(...); ...
glTexImage2D(...);
glEnable(GL_TEXTURE_2D);
```

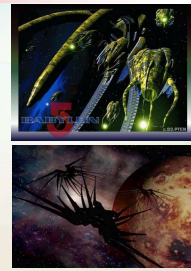
In display:

```
glBindTexture( ... ); // Activate the texture defined in initialization
glBegin(GL_TRIANGLES);
glTexCoord2f(...); glVertex3f(...);
glTexCoord2f(...); glVertex3f(...);
glTexCoord2f(...); glVertex3f(...);
glEnd();
```

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland
Computer Graphics NTUST CSIE Laboratory

Mapping, Eberly 2^E

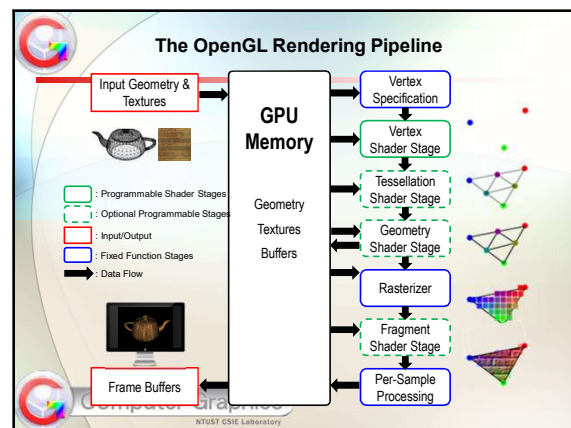
- Fine Surface Detail: **Bump** (§20.5 Eberly 2^E)
- Material Effects: **Gloss** (§20.6)
- Enclosing **Volumes**
 - Sphere (§20.7)
 - Cube (§20.8)
- Light
 - Refraction for **Transparency** (§20.9)
 - **Reflection** aka Environment (§20.10)
- Shadow
 - Shadow Maps (§20.11, 20.13)
 - Projective Textures (§20.12)
- More Special Effects (SFX)
 - Fog (§20.14)
 - Skinning (§20.15)
 - **Iridescence** (§20.16); **Water** (§20.17)



Babylon 5
© 1993 - 1998 Warner Brothers Entertainment, Inc.
Computer Graphics NTUST CSIE Laboratory

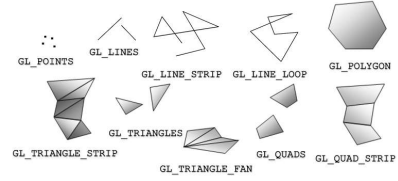
FOLLOWING THE PIPELINE

Computer Graphics
NTUST CSIE Laboratory

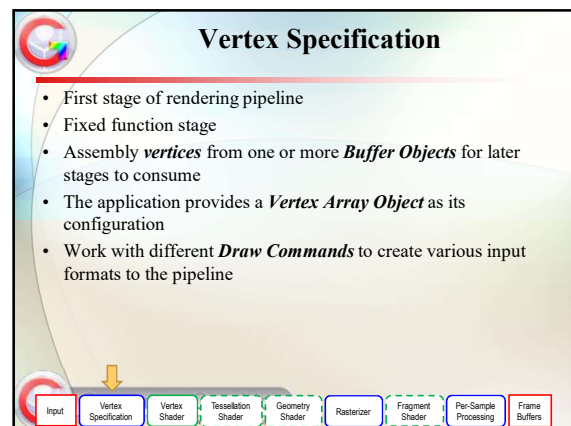


OpenGL Primitive

- Define the connection between the Vertex
- Setup when call the **Drawing Command**



Computer Graphics NTUST CSIE Laboratory



Vertex Specification: APIs

- This stage is mostly configured, but not all, by:
 - glBindVertexArray (vao)
 - glDraw* functions
 - Draw Command functions
 - These APIs also mark the beginning of a rendering operation
 - glDrawArrays, glDrawElements, ...

Vertex Shader Stage

- First **programmable** stage
- Invoked once for each vertex** of the vertex stream produced by the Vertex Specification
- Isolated from one another, **unaware of the primitive topology**

Vertex Shader Stage: Operations

- Apply transform matrices
- Vertex skinning (skeleton-based animations)
- Per-vertex lighting calculations
- Any operations related to the geometry (shape) of the rendered object

Vertex Shader Stage: Example

- Displacement Mapping: Vertex shader can offset every vertex based on a texture to have more detail.

Kenneth Scott, id Software 2008 copyright

Computer Graphics
NTUST CSIE Laboratory

Vertex Shader Stage: Example

- Water wave: Vertex shader can offset a 2D plane vertex to have different height according to time and location to create wave-like results.

NVIDIA water wave (<https://goo.gl/Pr7CgN>)

Computer Graphics
NTUST CSIE Laboratory

Tessellation Shader Stage

- Optional stage. Can be turned off
- Three sub-stages:
 - Tessellation Control Shader Stage
 - Tessellation Engine (fixed function stage)
 - Tessellation Evaluation Shader Stage
- Three sub-stages must be used together, or none at all

Tessellation Shader Stage

- Map **higher-order patches** that represent smooth, curved surfaces to conventional **triangle-based** raster hardware
 - Common misunderstanding: tessellation = triangulation (wrong)
- Demand for increasing image quality
 - High resolution models (1 character = 100K vertices)
 - Greatly increases on-disk, in-memory storage, I/O bandwidth and calculations
 - Scale geometry detail between different hardware from a fixed input
 - Relief the artists from participation in triangle-based mesh implementation details

Tessellation Shader Stage

- Map **higher-order patches** that represent smooth, curved surfaces to **conventional triangle-based** raster hardware
 - Common misunderstanding: **tessellation = triangulation (wrong)**
- Demand for **increasing image quality**
 - High resolution models (1 character = 100K vertices)
 - Greatly increases on-disk**, in-memory storage, I/O bandwidth and calculations
 - Scale geometry detail** between different hardware from a fixed input
 - Relief the artists from participation in triangle-based mesh** implementation details

Subdivide 1.00

Tessellation Shader Stage: Example

Tessellation Shader Stage: Example

- Terrain Generator : Subdivide the edge to present the curve with more detail.

NVIDIA Terrain generator Example (<https://goo.gl/RXYj9x>)

Tessellation Shader Stage: Example

Geometry Shader Stage

- The last stage that **can manipulate the geometry property** of the rendered object
- Invoked once for each primitive** from the previous stages
- Less common in algorithm designs, probably due to performance issues

G2 加一下郭鴻年的例子。
Graphics, 2017/9/15

Geometry Shader Stage: Operation

- Programmatically **insert/remove** geometry
- **Produce a different primitive type** than is passed to it
- Pass geometry information to buffers through the **transform feedback function**
 - **Transform Feedback**: save geometry to a buffer object from the geometry shader stage
 - Save results **generated by expensive vertex/tessellation shader** algorithms for cheap reuse
 - Useful when the GPU is used as a **co-processor to the CPU**. The GPU can process the geometry, then save the result for the CPU to read back

Geometry Shader Stage: Example

- (a) Geometry shader application : Normal Visualization
- (b) Geometry shader application : Layered Rendering

Computer Graphics
NTUST CSIE Laboratory

Geometry Shader Stage: Example

- (a) Geometry shader application : fish crowd simulation
- (b) Geometry shader application : fur simulation

NVIDIA fish crowd simulation (<https://goo.gl/3qg1WW>)
NVIDIA fur simulation (<https://goo.gl/xPwngn>)

Computer Graphics
NTUST CSIE Laboratory

Vertex Processing

- Vertex, Tessellation and Geometry Shader stages are also called **Vertex Processing**
- They all operate on vertices and geometries
- The last active stage of Vertex Processing stages should output a **clip space coordinate** to the Rasterizer stage

Rasterizer Stage

- Fixed function stage
- Consist of **three sub-stages**:
 1. Vertex **Post-Processing** Stage
 2. Primitive **Assembly** Stage
 3. **Rasterization** Stage
- Since they are not **really separated from one another**, and the hardware implementation **may differ from this order**, for convenience we will just discuss the functions they perform

Rasterizer Stage: Operations

- Operations performed, **in logical order**:
 1. Face Culling
 2. Primitive Culling
 3. Primitive Clipping (or Frustum Clipping)
 4. Perspective Division (or Homogeneous Division)
 5. Viewport Transformation
 6. Rasterization
 7. Multisampling

G3 加一下你自己列的例子。
Graphics, 2017/9/15

Rasterizer Stage: Operations

- Face Culling
 - Discard a primitive if it's facing towards or away the camera
 - Implement back face culling: for a watertight geometry, it is not possible to see its "inside", so back face will never contribute to the final rendering

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: Operations

- Primitive Culling
 - Discard a primitive if it's completely outside of the view volume
- Primitive Clipping (or Frustum Clipping)
 - If a primitive is partially inside the view volume, then it is split into new primitives that reside completely inside
 - Clip space coordinate make this process very efficient

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: Operations

- Perspective Division (or Homogeneous Division)
 - Transform the coordinate from clipping space to NDC
 - Covered in the previous lecture ☺
- Viewport Transformation
 - Transform the coordinate from NDC to screen space
 - Covered in the previous lecture ☺

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: Operations

- Rasterization
 - Convert the *geometric* data into a *regularly sampled representation* that can be written to *pixel-based images*, or be displayed on *pixel-based screens*

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: Operations

- Multisampling
 - A simplified, optimized and hardware-accelerated form of *anti-aliasing algorithm*
 - We will explain this in a future lecture ☺

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: APIs

- This stage is mostly configured, but not all, by:
 - Primitive Clipping
 - `glEnable/glDisable(GL_CLIP_DISTANCEi)`
 - Face Culling & Primitive Culling
 - `glEnable/glDisable(GL_CULL_FACE)`
 - `glFrontFace`
 - `glCullFace`
 - Perspective Division
 - None, cannot be controlled

↓

Input Vertex Specification Vertex Shader Tessellation Shader Geometry Shader Rasterizer Fragment Shader Per-Sample Processing Frame Buffers

Rasterizer Stage: APIs

- This stage is mostly configured, but not all, by:
 - Viewport Transformation
 - glViewport
 - glDepthRange
 - Rasterization
 - glPolygonMode
 - glEnable/glDisable (GL_POLYGON_OFFSET)
 - glPolygonOffset
 - glPointSize
 - glLineWidth
 - Multisampling
 - glEnable/glDisable (GL_MULTISAMPLE)

Fragment Shader Stage

- The last programmable stage
- Invoked once for each fragment** generated by the rasterizer
- Unaware of neighboring fragments.** Runs in parallel

Fragment Shader Stage: Operations

- In plain English: decide the color of a pixel
- Operate at a much higher frequency than the previous stages (1080p=2M pixels)
- Commonly used to add all the details to a rendering
- Perform lighting calculation and texturing operations

Fragment Shader Stage : Example

Original Ink painting Oil painting

Pixelize Invert Blur

Computer Graphics
NTUST CSIE Laboratory

Fragment Shader Stage : Example

- Fragment shader can control the object's visibility based on the distance from camera to simulate the fog effects.

Computer Graphics
NTUST CSIE Laboratory

Fragment Shader Stage : Example

Monet, Haystacks

Kitchen

Kitchen / Monet, Haystacks

Computer Graphics
NTUST CSIE Laboratory

Per-Sample Processing Stage

- The last stage **before a fragment** is written to the framebuffer
- Fixed function stage
- Perform three kind of tests:
 - Depth Test
 - Stencil Test
 - Scissor Test
- If a fragment passes all the tests, it will be written to the framebuffer. If **Blending** is enabled, blending is performed before the fragment is written

Per-Sample Processing Stage: Operations

- Scissor Test
 - Discard the fragments that are outside a user-specified rectangle
 - Useful for masking the output to a specific area

Per-Sample Processing Stage: Operations

- Stencil Test
 - Compare a reference value of incoming fragment with the contents of the **stencil buffer**
 - The content of the stencil buffer can also be updated dynamically
 - Like scissor test, useful for masking the output, but with more flexibility in algorithm design

Per-Sample Processing Stage: Operations

- Depth Test
 - Hardware implementation of the classical Z-buffer algorithm [Williams, 1978] with **depth buffer** to give the scene a correct depth order
 - Determines whether an incoming fragment can override or blend with the contents of the color buffer

Per-Sample Processing Stage: Operations

- Blending
 - Traditionally used for alpha-transparency rendering
 - Blend two color sources with a blending function. Sources and blending function are configurable

Per-Sample Processing Stage: APIs

- This stage is mostly configured, but not all, by:
 - Depth Test
 - `glEnable/glDisable(GL_DEPTH_TEST)`
 - `glDepthMask`
 - `glDepthFunc`
 - Stencil Test
 - `glEnable/glDisable(GL_STENCIL_TEST)`
 - `glStencilOp`
 - `glStencilFunc`
 - `glStencilMask`
 - Scissor Test
 - `glEnable/glDisable(GL_SCISSOR_TEST)`
 - `glScissor`
 - Blend
 - `glEnable/glDisable(GL_BLEND)`
 - `glBlendFunc`

投影片 116

G6 這幾個TEST你也有整理，也麻煩你把他們加入。
Graphics, 2017/9/16

G7 Graphics, 2017/9/16

投影片 117

G6 這幾個TEST你也有整理，也麻煩你把他們加入。
Graphics, 2017/9/16

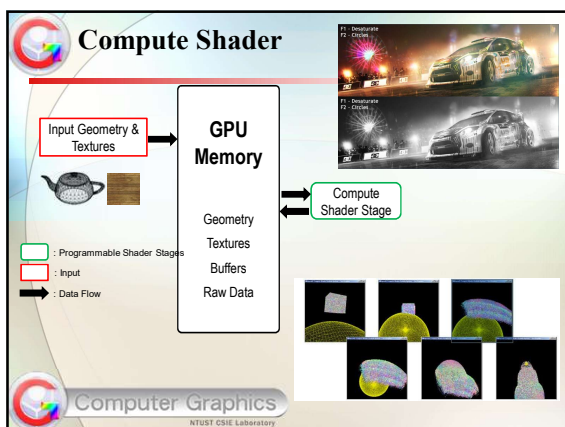
G7 Graphics, 2017/9/16

投影片 118

G8 這幾個TEST你也有整理，也麻煩你把他們加入。
Graphics, 2017/9/16

投影片 119

G8 這幾個TEST你也有整理，也麻煩你把他們加入。
Graphics, 2017/9/16



Compute Shader

- Available since OpenGL 4.3
- Compute shader pipeline only **has one programmable stage**
- There are **no predefined input or output** in compute shader stage. However, Compute shaders **can read/write buffers and textures asynchronously**, allowing for highly customized, parallel algorithm designs
- Idea close to **CUDA or DirectCompute**
- Compute shader is built upon the same foundation as the rendering pipeline stages, meaning it can **benefit from hardware texture filtering and GLSL intrinsic functions**

Computer Graphics
NTUST CSIE Laboratory

Compute Shader: Operations

- Compute shaders are becoming increasingly important in modern rendering algorithms
- **Physic-based simulations**
 - Water surface or cloth simulation
 - Particle system simulation
 - Audio reverb zone simulation
- **Procedural texture generation**
- **Image processing operations**
 - Separated 2D convolution (Gaussian blur, ...)
- GPGPU operations
 - Act as a co-processor to the CPU
 - General parallel algorithm designs

Computer Graphics
NTUST CSIE Laboratory

Compute Shader: Example

- (a) Compute shader application : water surface simulation
- (b) Compute shader application : particle simulation

Computer Graphics
NTUST CSIE Laboratory

Compute Shader: Operations

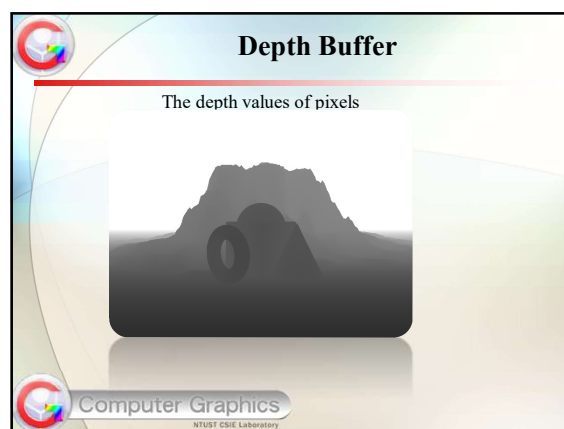
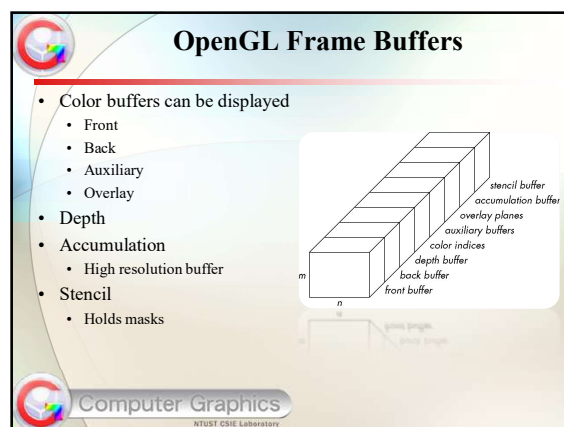
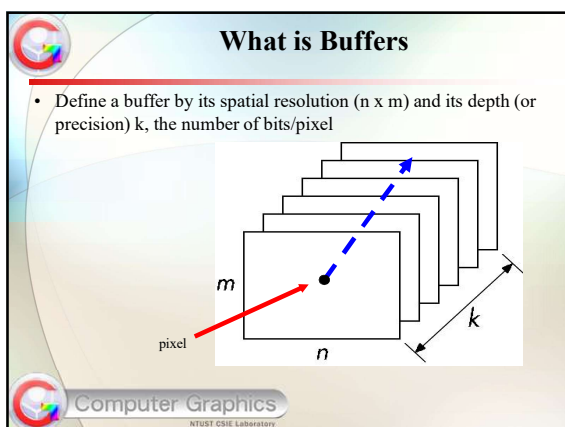
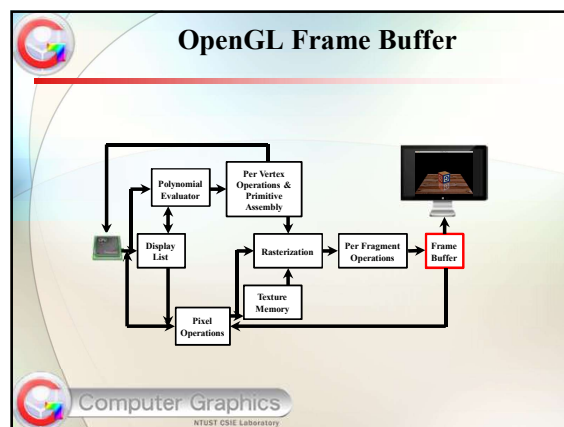
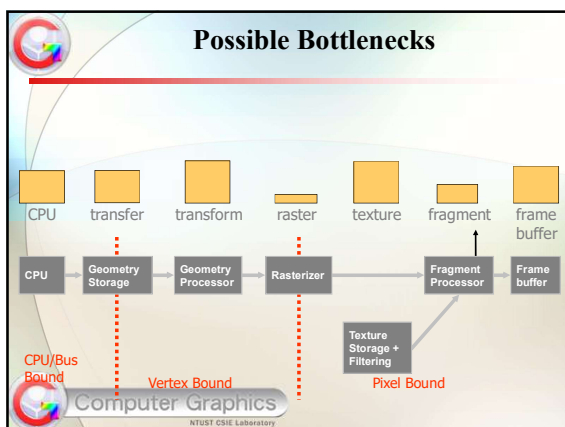
- Execute algorithmically general purpose GLSL shaders
 - Operate on uniforms, images, and textures.
- Process graphics data in the context of the graphics pipeline
 - Easier than interpolating with a compute API IF processing "close to the pixel"
- Complementary to OpenGL
 - Not a full heterogenous (CPU/GPU) programming framework using full ANSI C
- Standard part of all OpenGL 4.3 Implementations
 - Matches DirectX 11 Functionality

Computer Graphics
NTUST CSIE Laboratory

To think about:


- What are some possible **bottlenecks** in system performance of a graphics/game engine?
- Does it make any **difference to sort your geometry front-to-back or back-to-front** when using a depth-buffer?
- **Will your textured polygons render faster** if MIP-mapping is enabled or disabled?
- Does the order that you traverse polygons (i.e., issue vertices using `glVertex()`) make a difference in performance?

Computer Graphics
NTUST CSIE Laboratory David Luebke



Stencil Buffer

Stencil plays as a mask and indicates which pixels can be modified



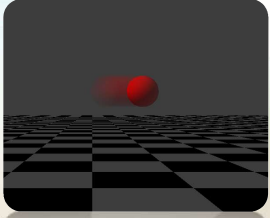
Without stencil

With stencil

Computer Graphics
NTUST CSIE Laboratory

Accumulation Buffer

- Accumulate information
 - Soft Shadows
 - Motion Blurs
 - Depth of Field



Computer Graphics
NTUST CSIE Laboratory

3D 遊戲設計專案



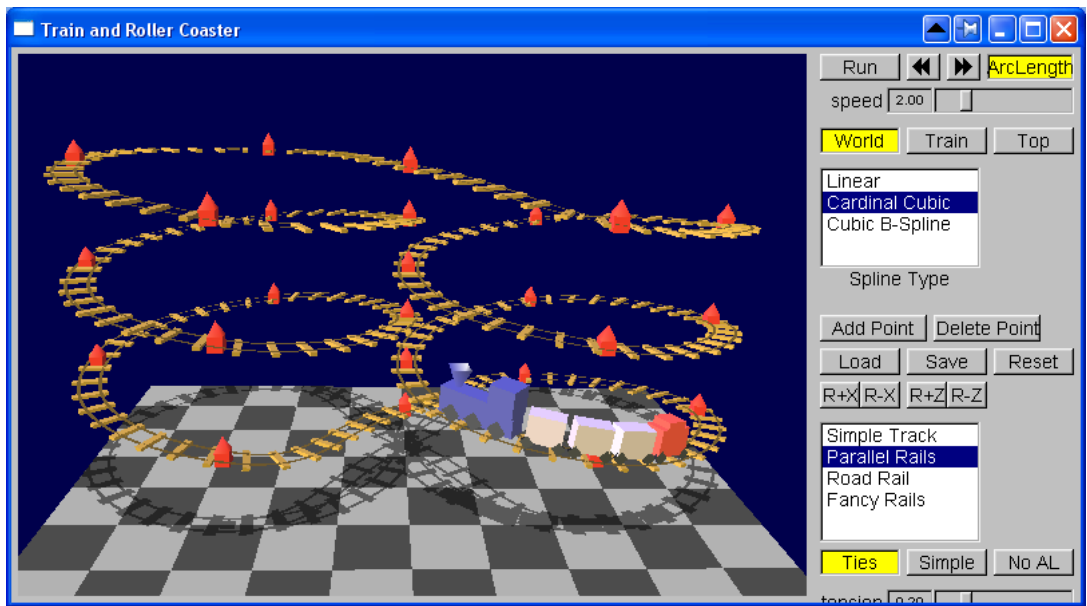
Menu

- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

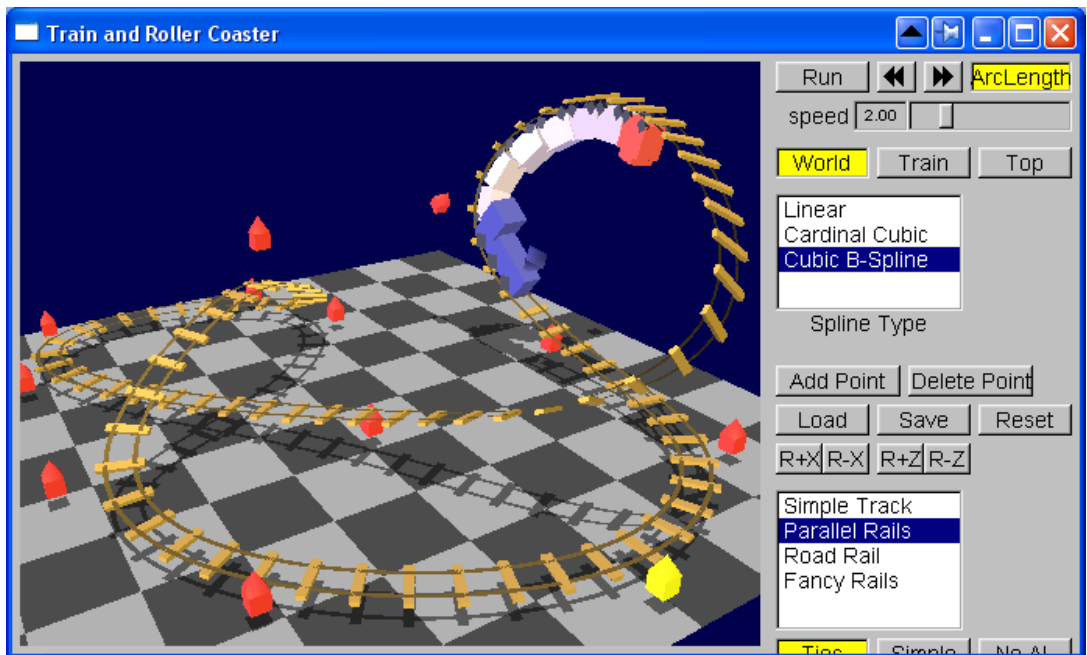
Project 1: Trains and Roller Coasters

Overview

In this project, you will create a train that will ride around on a track. When the track leaves the ground (or is very hilly), the train becomes more like a roller coaster.



Once it becomes a roller coaster, loops, corkscrews, and other things become possible



The main purposes of this project is to demonstrate your fundamental knowledge in Computer Graphics and to give you experience in working with a game engine (Unity, Unreal, or blender). The core of the project is a program that creates a 3D world, and to allow the user to place a train (or roller coaster) track in the world. This means that the user needs to be able to see and manipulate a set of control points that define a curve to represent the track, and that you can draw the track and animate the train moving along the track. It will also force you to use the game engine to create a fundamental interactive framework and proper user interfaces. The framework will be similar to the [framework](#) provided for Introduction to Computer Graphics and you can use it as a reference.

Requirements

Basically in this project you will need to:

- Create a framework as [this one](#) in your chosen game engine.
 1. Provide a user interface to look around the world, as well as providing a "top down" view.
 2. You must have a "ground" (so the track isn't just in space).
 3. Provide a user interface that allows control points to be added, removed, or repositioned.
 4. Allow for the control points to be saved and loaded from text files in the format used by the example solution.
 5. Provide lighting.
 6. Allow things to be animated (have a switch that allows the train to start/stop), as well as allowing for manually moving the train forward and backwards.
 7. You should have a slider (or some control) that allows for the speed of the train to be adjusted (how far the train goes on each step, not the number of steps per second).
- Add the basic functionality:
 - Draw a track based on the control points.
 1. Rail ties: ties are the cross pieces on railroad tracks. Getting them right (uniformly spaced) requires good arc-length parameterization. In the example code, you can turn the arc-length parameterization on and off to see the difference.
 2. Arc-length parameterization: Having your train move at a constant velocity (rather than moving at a constant change of parameter value) makes things better. Implementing this is an important step towards many other advanced features. You should allow arc-length parameterization to be switched on and off to emphasize the difference, you should also provide a speed control.
 3. Control the orientation of track: the simple schemes for orienting the train break down in 3D - in particular, when there are loops. Make it so that your train consistently moves along the track (so its under the track at the top of a loop).
 - One good way to provide for proper orientations is to allow the user to control which direction is "up" at points along the curve. This allows you to do things like corkscrew roller coasters. The sample solution does this (its why the framework has an orientation vector for each control point). Note that the train still needs to face forward, the given orientation is just a hint as to which way up should be.
 - Simple physics: Roller coasters do not go at constant velocities - they speed up and slow down. Simulating this (in a simple way) is really easy once you have arc-length parameterization. Remember that Kinetic Energy - Potential Energy should remain constant (or decrease based on friction). This lets you compute what the velocity should be based on how high the roller coaster is. Even Better is to have "Roller Coaster Physics" - the roller coaster is pulled up the first hill at a constant velocity, and "dropped " where it goes around the track in "free fall." You could even have it stop at the platform to pick up people. Note: you should implement arc-length first. Once you get it right it is much easier.
 - Draw a train on that track.
 1. Have the train oriented correctly on the track.
 2. Have a rider: Little people who put their hands up as they accelerate down the hill are a cool addition. (I don't know why putting your hands up makes roller coasters more fun, but it does). The hands going up when the train goes down hill is a requirement.
 3. Headlight for the Train: Have the train have a headlight that actually lights up the objects in front of it. This is actually very tricky since it requires local lighting, which isn't directly supported.
 4. Multiple cars
 5. Have Real Train Wheels: Real trains have wheels at the front and back that are both on the track and that swivel relative to the train itself. If you make real train wheels, you'll need arc-length parameterization to keep the front and rear wheels the right distances apart (make sure to draw them so we can see them swiveling when the train goes around a tight turn).

In the sample solution, the wheels are trucked (they turn independently), but each car still rotates around its center (so its as if they are floating above the wheels). You can do better than that.

- Add advanced features
 1. Add shadow: shadow gives the user sense of space. It is important to add shadow in the interactive graphics applications.
 2. Non-Flat Terrain for the ground: This is mainly interesting if you have the train track follow the ground (maybe with tressles or bridges if the ground is too bumpy).
 3. Have the train make smoke: Steam trains are the coolest trains, even if they are being a roller coaster. Having some kind of smoke coming from the train's smoke stack would be really neat. Animate the smoke (for example, have "balls of smoke" that move upward and dissipate).
 4. Add sky with sky box, sky plane, sky sphere, and etc.
 5. Load in models created by blender, Maya, 3DMax, and other 3D tools.
 6. Scenery: having other (non-moving) objects in the world gives you something to look at when you ride the train.
 7. 3 Shaders: Graphics Processing Unit(GPU) has become very important aspect in graphics. We would like to explore the usage of them such as environment map, particle simulation, water simulation, and so on. Everyone has to write 3 shaders

Grading Criterion

This project is similar to those in [Project 3](#) and [Project 4](#) in Introduction to Computer Graphics. Therefore, the score is the advance features in Project 3 and Project4 without the water feature (we will implement it in P2). Here is the grading sheet.

What to handin

- a readme.txt which should explain the following:
 1. A list of all the features that you have added, including a description, and an explanation of how you know that it works correctly.
 2. A discussion of any important, technical details (like how you compute the coordinate system for the train, or what method you use to compute the arc length)
 3. Anything else we should know to compile and use your progra
- 3 screen shots
- Source code and media information
- A 1~2 minutes video



enu

Home

Faculty

Students

Projects

• Research

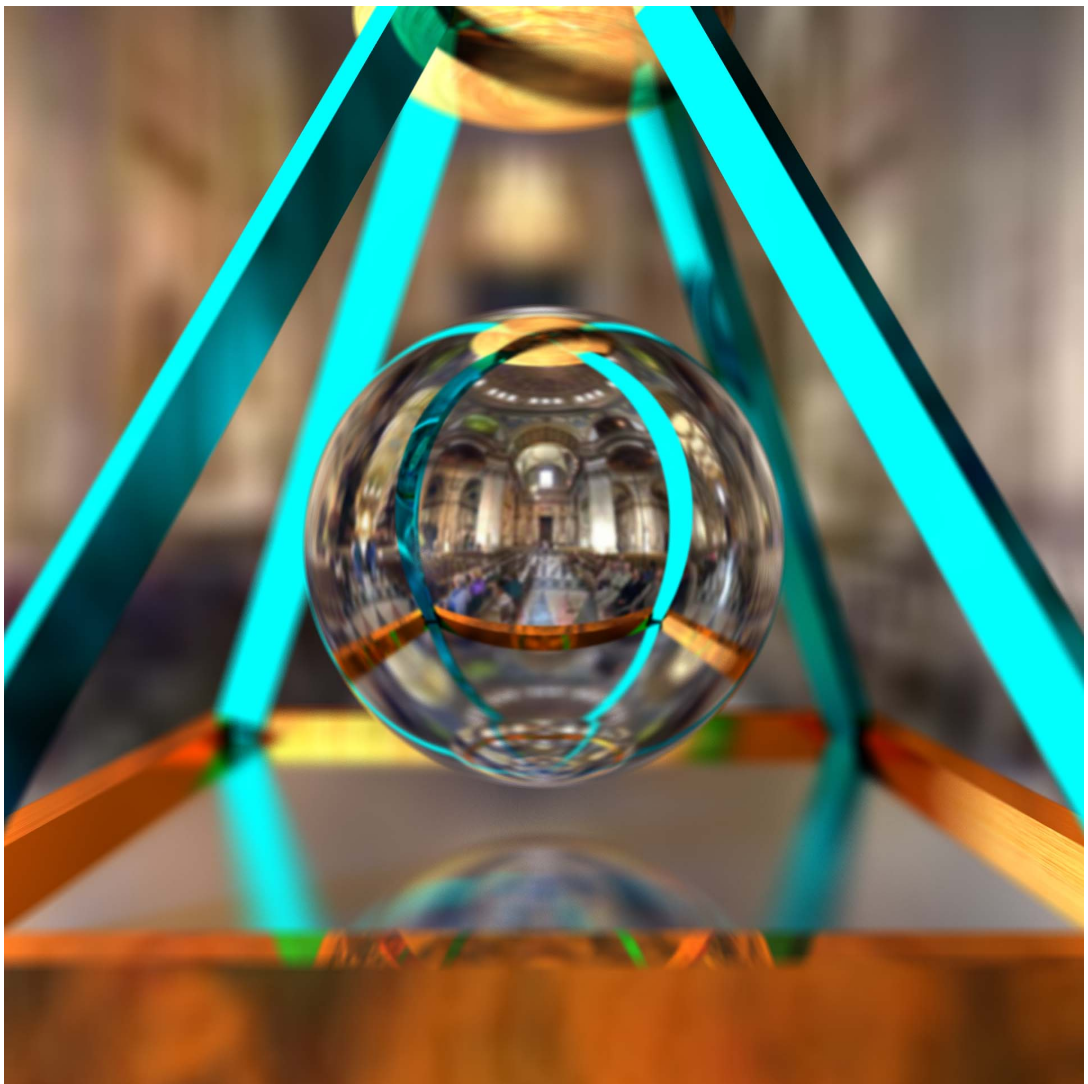
• Games

• Others

Courses

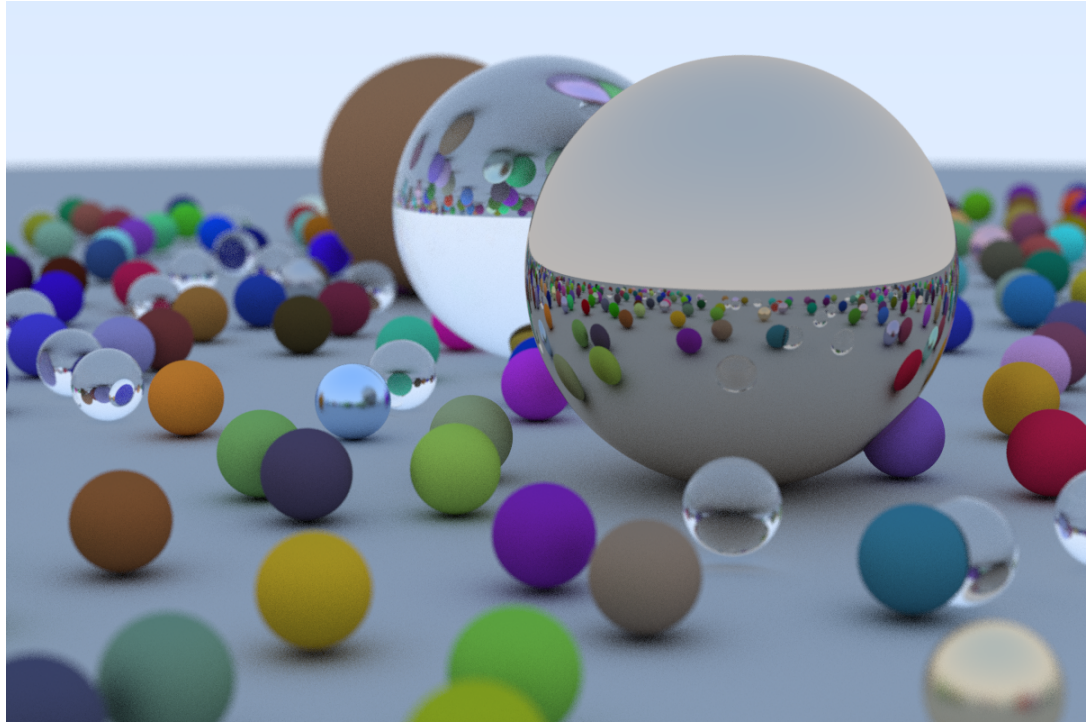
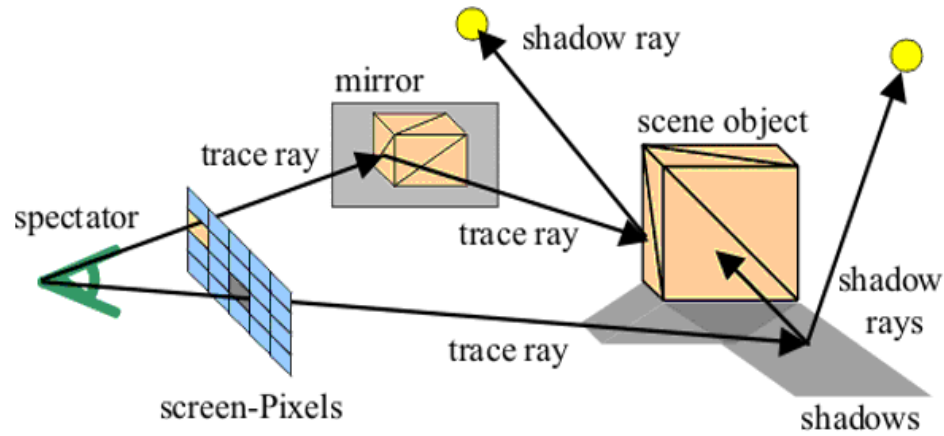
Project 2: Ray Tracer

Introduction



In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration).

In this project, you will build a program called Ray that will generate ray-traced images of complex scenes. The ray tracer should trace rays recursively using the Whitted illumination model.



Algorithm Overview

Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than either ray casting or scanline rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it.

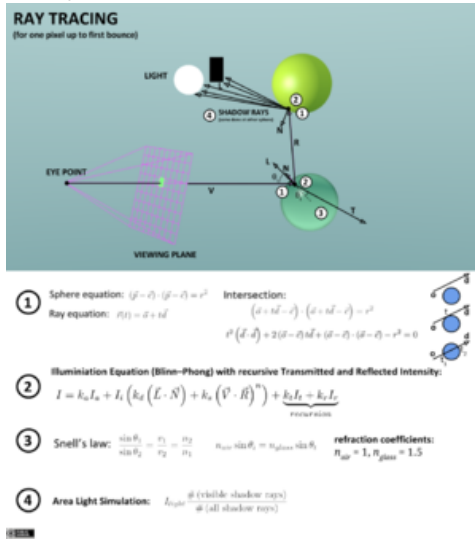
Scenes in ray tracing are described mathematically by a programmer or by a visual artist (typically using intermediary tools). Scenes may also incorporate data from images and models captured by means such as digital photography.

Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene.

It may at first seem counterintuitive or "backwards" to send rays away from the camera, rather than into it (as actual light does in reality), but doing so is many orders of magnitude more efficient. Since the overwhelming majority of light rays from a given light source do not make it directly into the viewer's eye, a "forward" simulation could potentially waste a tremendous amount of computation on light paths that are never recorded.

Therefore, the shortcut taken in raytracing is to presuppose that a given ray intersects the view frame.

After either a maximum number of reflections or a ray traveling a certain distance without intersection, the ray ceases to travel and the pixel's value is updated.



Pros and Cons

- Advantages over other rendering methods:
 - Ray tracing's popularity stems from its basis in a realistic simulation of **lighting** over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and **shadows**, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. The computational independence of each ray makes ray tracing amenable to **parallelization**.
- Disadvantages:
 - A serious disadvantage of ray tracing is performance (while it can in theory be faster than traditional scanline rendering depending on scene complexity vs. number of pixels on-screen). Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform **spatial anti-aliasing** and improve image quality where needed.
 - Although it does handle interreflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the **rendering equation** is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required. The realism of all rendering methods can be evaluated as an approximation to the equation. Ray tracing, if it is limited to Whitted's algorithm, is not necessarily the most realistic. Methods that trace rays, but include additional techniques (**photon mapping**, **path tracing**), give far more accurate simulation of real-world lighting.

The Basic Task

```

for every pixel {
    cast a ray from the eye
    for every object in the scene {
        find intersections with the ray keep it if closest
    }
    compute color at the intersection point
}
    
```

Required Functionality

We'll describe these requirements in more detail afterwards:

1. Parse in the scene, material, light and other informations. (5%)
 2. Fast local rendering with OpenGL. (5%)
 3. Generate the tracing rays from the camera. (5%)
 4. Use the intersection mechanism in Engine for ray-object intersection computation. (5%)
 5. Implement the Whitted illumination model, which includes Phong *shading* (emissive, ambient, diffuse, and specular terms) as well as reflection and refraction terms. You only need to handle directional and point light sources, i.e. no area lights, but you should be able to handle multiple lights. (20%)
 6. Implement Phong *interpolation* of normals on triangle meshes. (10%)
 7. Implement anti-aliasing. Regular super-sampling is acceptable, more advanced anti-aliasing will be considered as an extension. (10%)
 8. Implement data structures that speed up the intersection computations in large scenes. There will be a contest at the end of the project to determine the team with the fastest ray tracer. (10%)
- Notes on Whitted's illumination model

The first three terms in Whitted's model will require you to trace rays towards each light, and the last two will require you to recursively trace reflected and refracted rays. (Notice that the number of reflected and refracted rays that will be calculated is limited by the "depth" setting in the ray tracer. This means that to see reflections and refraction, you must set the depth to be greater than zero!)

When tracing rays toward lights, you should look for intersections with objects, thereby rendering shadows. If you intersect a semi-transparent object, you should attenuate the light, thereby rendering partial (color-filtered) shadows, but you may ignore refraction of the light source. The skeleton code doesn't implement Phong interpolation of normals. You need to add code for this (only for meshes with per-vertex normals.)

[Here](#) is a document that lists equations that will come in handy when writing your shading and ray tracing algorithms.
 - Anti-aliasing

Once you've implemented the shading model and can generate images, you will notice that the images you generated are filled with "jaggies". You should implement an anti-aliasing technique to smooth these rough edges. In particular, you are required to perform super-sampling and averaging down. You should provide a slider which allows the user to control the number of samples per pixel (1, 4, 9 or 16 samples). You need only implement a box filter for the averaging down step. More sophisticated anti-aliasing methods are left as bells and whistles below.
 - Accelerated ray-surface intersection

The goal of this portion of the assignment is to speed up the ray-surface intersection module in your ray tracer. In particular, we want you to improve the running time of the program when ray tracing complex scenes containing large numbers of objects (they are usually triangles). There are two basic approaches to do this:

 1. Specialize and optimize the ray-object intersection test to run as fast as possible.
 2. Add data structures that speed the intersection query when there are many objects.

Most of your effort should be spent on approach 2, i.e. reducing the number of ray-object intersection tests. You are free to experiment with any of the acceleration schemes described in Chapter 6, "A Survey of Ray Tracing Acceleration Techniques," of Glassner's book. Of course, you are also free to invent new acceleration methods.

Make sure that you design your acceleration module so that it is able to handle the current set of geometric primitives - that is, triangles spheres, squares, boxes, and cones.

The sample scenes include several simple scenes and three complex test scenes: trimesh1, trimesh2, and trimesh3. You will notice that trimesh1 has per-vertex normals and materials, and trimesh2 has per-vertex materials but not normals. Per-vertex normals and materials imply interpolation of these quantities at the current ray-triangle intersection point (using barycentric coordinates).

What to hand in?

All your hand-in must be put in a directory with your student ID and the following is the list of hand-in files under the directory.

- Program and source: As usual, you must hand in everything needed to build and run your program, including all texture files and other resources.
- Gallery: Please put a few JPG pictures of the rendering results at least three. Please name the pictures ID-X.jpg (where X is a number).
- Read-me.txt:
 - Instructions on how to use your program (in case we want to use it when you're not around)

- Descriptions of what your program does to meet all of the minimum requirements.
- Technical.txt:
 - The report could contain a description of this project, what you have learned from this project, description of the algorithm you implemented, implementation details, results (either good or bad), and what extensions you have implemented.

Extra Effects

Here are some [examples](#) of effects you can get with ray tracing. Currently none of these were created from past students' ray tracers.

- Implement an adaptive termination criterion for tracing rays, based on ray contribution. Control the adaptation threshold with a slider.
- Implement stochastic (jittered) supersampling. See Glassner, Chapter 5, Section 4.1 - 4.2 and the first 4 pages of Section 7
- Modify shadow attenuation to use Beer's law, so that the thicker objects cast darker shadows than thinner ones with the same transparency constant. (See Shirley p. 214.)
- Include a Fresnel term so that the amount of reflected and refracted light at a transparent surface depend on the angle of incidence and index of refraction. (See Shirley p. 214.)
- Add a menu option that lets you specify a background image to replace the environment's ambient color during the rendering. That is, any ray that goes off into infinity behind the scene should return a color from the loaded image, instead of just black. The background should appear as the backplane of the rendered image with suitable reflections and refractions to it.
- Deal with overlapping objects intelligently. In class, we discussed how to handle refraction for non-overlapping objects in air. This approach breaks down when objects intersect or are wholly contained inside other objects. Add support to the refraction code for detecting this and handling it in a more realistic fashion. Note, however, that in the real world, objects can't coexist in the same place at the same time. You will have to make assumptions as to how to choose the index of refraction in the overlapping space. Make those assumptions clear when demonstrating the results.
- Implement spot lights.
- Implement antialiasing by adaptive supersampling, as described in Glassner, Chapter 1, Section 4.5 and Figure 19 or in Foley, et al., 15.10.4. For full credit, you must show some sort of visualization of the sampling pattern that results. For example, you could create another image where each pixel is given an intensity proportional to the number of rays used to calculate the color of the corresponding pixel in the ray traced image. Implementing this bell/whistle is a big win -- nice antialiasing at low cost.
- Add some new types of geometry to the ray tracer. Consider implementing torii or general quadrics. Many other objects are possible here.
- Implement more versatile lighting controls, such as the Warn model described in Foley 16.1.5. This allows you to do things like control the shape of the projected light.
- Implement stochastic or distributed ray tracing to produce one or more of the following effects: depth of field, soft shadows, motion blur, glossy reflection (See Glassner, chapter 5, or Foley, et al., 16.12.4).
- Add texture mapping support to the program. To get full credit for this, you must add uv coordinate mapping to all the built-in primitives (sphere, box, cylinder, cone) except trimeshes. The square object already has coordinate mapping implemented for your reference. The most basic kind of texture mapping is to apply the map to the diffuse color of a surface. But many other parameters can be mapped. Reflected color can be mapped to create the sense of a surrounding environment. Transparency can be mapped to create holes in objects. Additional (variable) extra credit will be given for such additional mappings. The basis for this bell is built into the skeleton, and the parser already handles the types of mapping mentioned above. Additional credit will be awarded for quality implementation of texture mapping on general trimeshes.
- Implement bump mapping (Watt 8.4; Foley, et al. 16.3.3). Check [this out!](#)
- Implement solid textures or some other form of procedural texture mapping, as described in Foley, et al., 20.1.2 and 20.8.3. Solid textures are a way to easily generate a semi-random texture like wood grain or marble.
- Extend the ray-tracer to create Single Image Random Dot Stereograms (SIRDS). [Click here](#) to read a paper on how to make them. Also check out this [page of examples](#). Or, create 3D images like [this one](#), for viewing with red-blue glasses.
- Implement 3D fractals and extend the [.ray file format](#) to provide support for these objects. Note that you are not allowed to "fake" this by just drawing a plain old 2D fractal image, such as the usual Mandelbrot Set. Similarly, you are not allowed to cheat by making a .ray file that arranges objects in a fractal pattern, like the sier.ray test file. You must raytrace an actual 3D fractal, and

your extension to the .ray file format must allow you to control the resulting object in some interesting way, such as choosing different fractal algorithms or modifying the base pattern used to produce the fractal. Here are two really good examples of raytraced fractals that were produced by students during a previous quarter: [Example 1](#), [Example 2](#). And here are a couple more interesting fractal objects: [Example 3](#), [Example 4](#)

- Implement 4D quaternion fractals and extend the .ray file format to provide support for these objects. These types of fractals are generated by using a generalization of complex numbers called quaternions. What makes the fractal really interesting is that it is actually a 4D object. This is a problem because we can only perceive three spatial dimensions, not four. In order to render a 3D image on the computer screen, one must "slice" the 4D object with a three dimensional hyperplane. Then the points plotted on the screen are all the points that are in the intersection of the hyperplane and the fractal. Your extension to the .ray file format must allow you to control the resulting object in some interesting way, such as choosing different generating equations, changing the slicing plane, or modifying the surface attributes of the fractal. Here are a few examples, which were created using the [POV-Ray raytracer](#) (yes, POV-Ray has quaternion fractals built in!): [Example 1](#), [Example 2](#), [Example 3](#), [Example 4](#). And, [this](#) is an excellent example from a previous quarter. To get started, visit this web page to brush up on your [quaternion math](#). Then [go to this site](#) to learn about the theory behind these fractals. Then, you can take a look at [this page](#) for a discussion of how a raytracer can perform intersection calculations.
- Implement a more realistic shading model. Credit will vary depending on the sophistication of the model. A simple model factors in the Fresnel term to compute the amount of light reflected and transmitted at a perfect dielectric (e.g., glass). A more complex model incorporates the notion of a microfacet distribution to broaden the specular highlight. Accounting for the color dependence in the Fresnel term permits a more metallic appearance. Even better, include anisotropic reflections for a plane with parallel grains or a sphere with grains that follow the lines of latitude or longitude. Sources: Shirley, Chapter 24, Watt, Chapter 7, Foley et al, Section 16.7; Glassner, Chapter 4, Section 4; Ward's SIGGRAPH '92 paper; Schlick's Eurographics Rendering Workshop '93 paper. This all sounds kind of complex, and the physics behind it is. But the coding doesn't have to be. It can be worthwhile to look up one of these alternate models, since they do a much better job at surface shading. Be sure to demo the results in a way that makes the value added clear. Theoretically, you could also invent new shading models. For instance, you could implement a *less* realistic model! Could you implement a shading model that produces something that looks like cel animation? Variable extra credit will be given for these "alternate" shading models. Links to ideas: [Comic Book Rendering](#). Note that you must still implement the Phong model.
- Implement CSG, constructive solid geometry. This extension allows you to create very interesting models. See page 108 of Glassner for some implementation suggestions. An [excellent example of CSG](#) was built by a grad student here in the grad graphics course.
- Add a particle systems simulation and renderer (Foley 20.5, Watt 17.7, or see instructor for more pointers).
- Implement caustics by tracing rays from the light source and depositing energy in texture maps (a.k.a., illumination maps, in this case). Caustics are variations in light intensity caused by refractive focusing--everything from simple magnifying-glass points to the shifting patterns on the bottom of a swimming pool. [A paper](#) discussing some methods. 2 bells each for refractive and reflective caustics. (Note: caustics can be modeled without illumination maps by doing "photon mapping", a monster bell described below.) Here is a really good example of caustics that were produced by two students during a previous quarter: [Example](#)

Advance Technology

There are innumerable ways to extend a ray tracer. Think about all the visual phenomena in the real world. The look and shape of cloth. The texture of hair. The look of frost on a window. Dappled sunlight seen through the leaves of a tree. Fire. Rain. The look of things underwater. Prisms. Do you have an idea of how to simulate this phenomenon? Better yet, how can you *fake* it but get something that looks just as good? You are encouraged to dream up other features you'd like to add to the base ray tracer. Obviously, any such extensions will receive variable extra credit depending on merit (that is, coolness!). Feel free to discuss ideas with the course staff before (and while) proceeding!

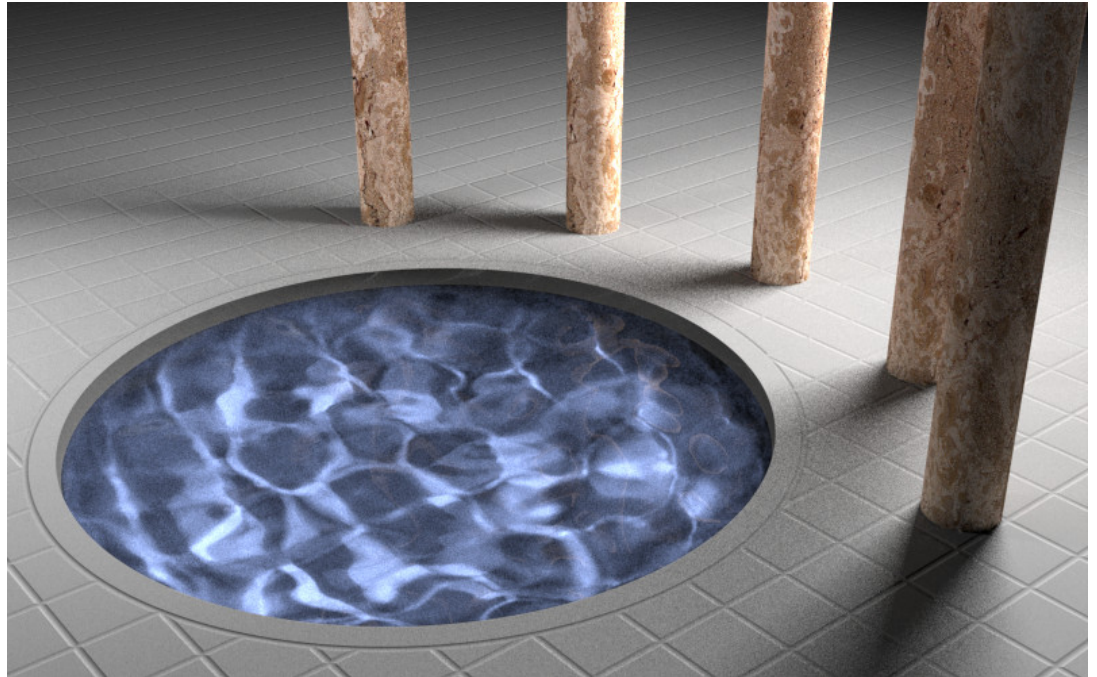
- Sub-Surface Scattering



The trace program assigns colors to pixels by simulating a ray of light that travels, hits a surface, and then leaves the surface at the same position. This is good when it comes to modeling a material that is metallic or mirror-like, but fails for translucent materials, or materials where light is scattered beneath the surface (such as skin, milk, plants...).

Check [this](#) paper out to learn more.

- Metropolis Light Transport



Not all rays are created equal. Some light rays contribute more to the image than others, depending on what they reflect off of or pass through on the route to the eye. Ideally, we'd like to trace the rays that have the largest effect on the image, and ignore the others. The problem is: how do you know which rays contribute most? Metropolis light transport solves this problem by randomly searching for "good" rays. Once those rays are found, they are mutated to produce others that are similar in the hope that they will also be good. The approach uses statistical sampling techniques to make this work. Here's some [information](#) on it, and a neat [picture](#).

- Photon Mapping



Photon mapping is a powerful variation of ray tracing that adds speed, accuracy and versatility. It's a two-pass method: in the first pass photon maps are created by emitting packets of energy (photons) from the light sources and storing these as they hit surfaces within the scene. The scene is then rendered using a distribution ray tracing algorithm optimized by using the information in the photon maps. It produces some amazing pictures. Here's some [information](#) on it.

Reference

- General
 1. *An Improved Illumination Model for Shaded Display*, T. Whitted, CACM, 1980, pp 343-349
 2. *An Introduction to Ray Tracing*, Andrew S. Glassner. (Chap. 6 for acceleration)
- Space Subdivision

1. *Ray Tracing with the BSP Tree*, K Sung & P. Shirley. [Graphics Gems III](#).
 2. *ARTS: Accelerated Ray-Tracing System*, A. Fujimoto et. al. *CG&A* April 1986, pp 16-25.
 3. *A Fast Voxel Traversal Algorithm for Ray Tracing*, J. Amanatides & A. Woo. *Eurographics'87*, pp 3-9
 4. *Faster Ray Tracing Using Adaptive Grids*, K. Klimaszewski & T. Sederberg. *CG&A* Jan. 1997, pp 42-51 (It is claimed to be the fastest algorithm so far.)
- Hierarchical Bounding Volume
 1. *Automatic Creation of Object Hierarchies for Ray Tracing*, J. Goldsmith & J. Salmon. *CG&A* May 1987, pp 14-20.
 2. *Efficiency Issues for Ray Tracing*, B. Smits. *Journal of Graphics Tools*, Vol. 3, No. 2, pp. 1-14, 1998.
 3. *Ray Tracing News*, Vol. 10, No. 3.
 - Tips
 1. *Fast Ray-Box Intersection*, A. Wu [Graphics Gems](#).
 2. *Improved Ray Tagging for Voxel-Based Ray Tracing*, D. Kirk & J. Arvo. [Graphics Gems](#). II
 3. *Rectangular Bounding Volumes for Popular Primitives*, B. Trumbore. [Graphics Gems](#). III
 4. *A Linear-Time Simple Bounding Volume Algorithm*, X. Wu. [Graphics Gems](#). III
 5. *A Fast Alternative to Phong's Specular Model*, Christophe Schlick [Graphics Gems](#) IV.
 6. *Voxel Traversal along a 3D Line*, D. Cohen. [Graphics Gems](#). IV.
 7. *Faster Refraction Formula, and Transmission Color Filtering*, *Ray Tracing News*, Vol. 10, No. 1.

Copyright © 2019 NTUST CSIE Computer Graphics Lab. All right reserved.



Menu

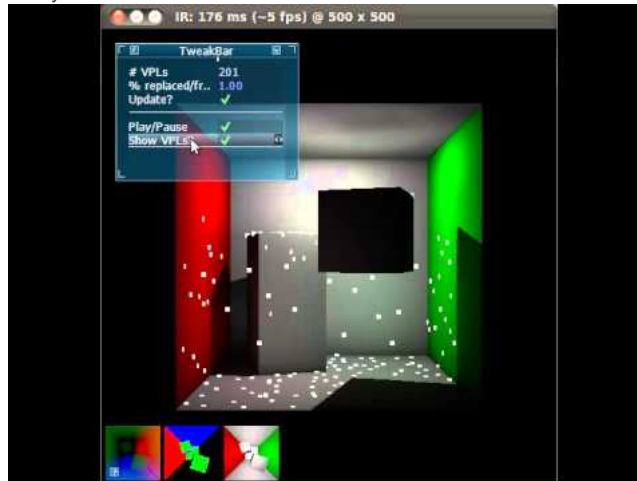
- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 3: Incremental Instant Radiosity Implementation

Introduction

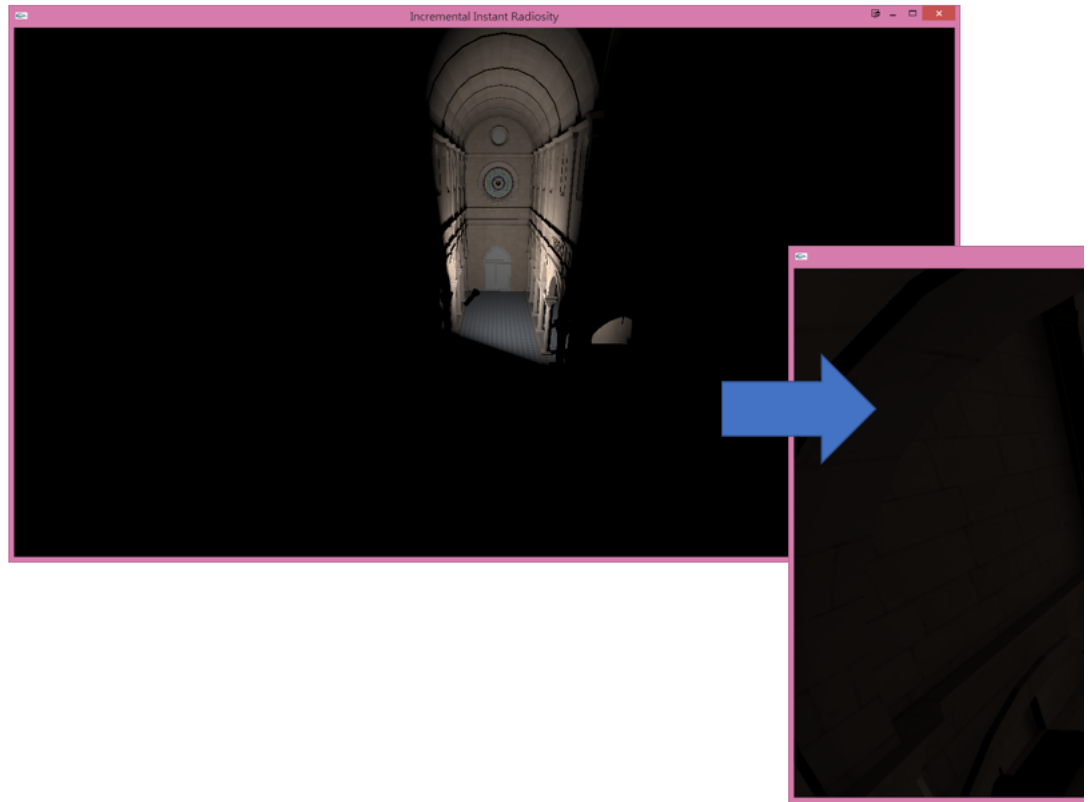


Instant radiosity is important techniques for real-time global illumination which is a fundamental procedure for instant rendering from the radiance equation. Operating directly on the textured scene description, the very efficient and simple algorithm produces photorealistic images without any finite element kernel or solution discretization of the underlying integral equation. However, its efficiency is still limited. In addition to regenerating all point light sources, incremental instant radiosity is proposed for rendering single-bounce indirect illumination in real time on currently available graphics hardware. The method is based on the instant radiosity algorithm, where virtual point lights (VPLs) are generated by casting rays from the primary light source. Hardware shadow maps are then employed for determining the indirect illumination from the VPLs. Our main contribution is an algorithm for reusing the VPLs and incrementally maintaining their good distribution. As a result, only a few shadow maps need to be rendered per frame as long as the motion of the primary light source is reasonably smooth. This yields real-time frame rates even when hundreds of VPLs are used.

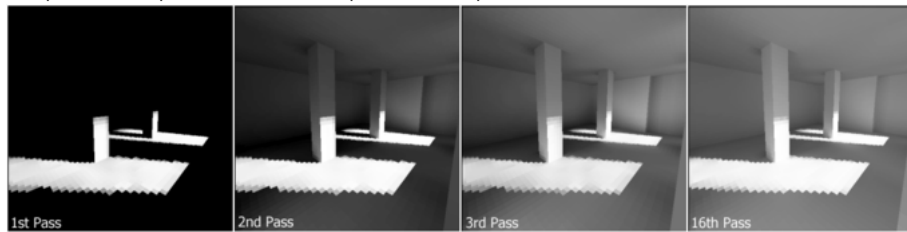


Simple Algorithm Overview

Instant radiosity process: For a full description of the algorithm, see the original paper.... However, here is a (very) brief overview. At runtime, a number of photons are selected from the light sources to be cast into the scene. In addition, the mean reflectivity r of the scene is calculated. Initially there are N photons. For each photon, the scene is rendered, placing a point light source at the origin of the photon. Then, rN of these photons are cast into the scene. When a photon hits a surface, it is attenuated by the diffuse component of that surface. Then the scene is rendered again, with the point light source appropriately moved. r^2N of the original points continue on to a second bounce.

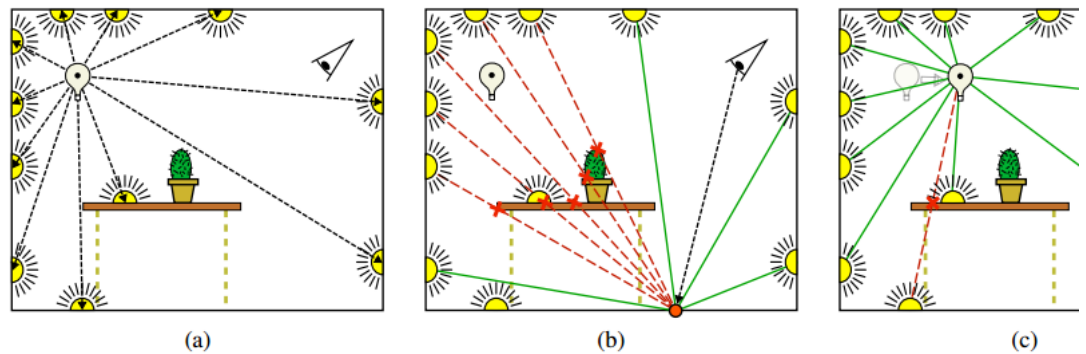


The process is repeated until all of the paths are complete.



In essence, this algorithm is generating an increasingly accurate approximation of the average radiance for every point in the scene with the rendering pass.

Incremental process:



You must first justify the existence of each VPLs because the generation rays may be occluded due to the movement of the light sources and objects. You must remove all invalid VPLs. You must also adjust the contribution of valid VPLs. Furthermore, you must increase the VPLs to the desired numbers. Finally, you render the scene with all existing VPLs.

Pros and Cons

The advantages and disadvantages of the incremental instant radiosity is as follows:

Significant advantages:

- Can use OpenGL hardware to decrease rendering time
- Computed solution can be displayed directly
- Low memory requirements, since it works in image space rather than creating matrix elements
- Algorithm can be extended to allow specular surfaces
- Radiance contribution from textures is directly computed

Disadvantages:

- View dependent (although Keller does present a method of generating walk-throughs in the paper)
- This method is not terribly accurate, it generates nice pictures, but should not be used for predictive results.

- Final output quality is dependant on hardware capabilities. For instance, the accumulation buffer needs to be fairly deep to allow large numbers of images to be composited.
- Doesn't work as well on scenes lit primarily by indirect lighting. Since it uses a quasi-random walk to distribute photons, several hundred frames may be required to get photons to arrive where they are needed.

The Basic Task

Radiosity is a global illumination algorithm that handles diffuse interreflections between surfaces.

$$L(y, \vec{\omega}_r) = L_e(y, \vec{\omega}_r) + \int_{\Omega} f_r(\vec{\omega}_i, y, \vec{\omega}_r) L(h(y, \vec{\omega}_i), -\vec{\omega}_i)$$

Instant Radiosity uses a Quasi-Monte Carlo approach to solve this integral and creates point light sources at each bounce for rays cast from the light source. If the light sources and resulting bounces are sampled adequately, this yields a good approximation of the global lighting in the scene. The core algorithm (with indicated modifications) is as follows:

```
void InstantRadiosity(int N, double p̄)
{
    double w, Start; int End, Reflections = 0;
    Color L; Point y; Vector ω;

    Start = End = N;

    while(End > 0)
    {
        Start *= p̄;

        for(int i = (int) Start; i < End; i++)
        {
            // Select starting point on light source
            y = y0(Φ2(i), Φ3(i));
            L = Le(y) * supp Le;
            w = N;

            // trace reflections
            for(int j = 0; j <= Reflections; j++)
            {
                Create light source with intensity L/[w]

                // diffuse scattering
                ω = ωd(Φb2j+2(i), Φb2j+3(i));
                //trace ray from y into direction ω
                y = h(y, ω);
                // Attenuate and compensate
                L *= fd(y) * AttenuationFactor;
                w *= p̄;
            }

            Reflections++;
            End = (int) Start;
        }
    }
}
```

Required Functionalities

1. Use P1's parser to parse in the scene, material, light and other informations.
2. Fast local rendering with OpenGL.
3. Shoot out light rays from light sources to generate VPLs. (5%)
4. Visualize these VPLs with OpenGL. (5%)
5. Generate a shadow map for each VPL. (10%)
6. Rendering the scene by rasterizing all scene triangles by shading with all VPLs with their shadow map. (10%)

7. Dynamics updating the characteristics of VPLs with incremental instant radiosity. (25%)
 1. Remove invalid VPLs.
 2. Adjust valid VPLs.
 3. Add new VPLs.
8. Acceleration with KD-tree or BVH (15%)

What to Hand in

All your hand-in must be put in a directory with your student ID and the following is the list of hand-in files under the directory.

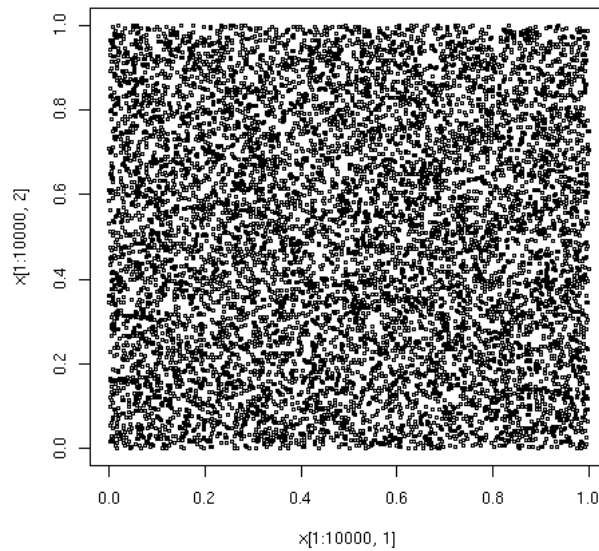
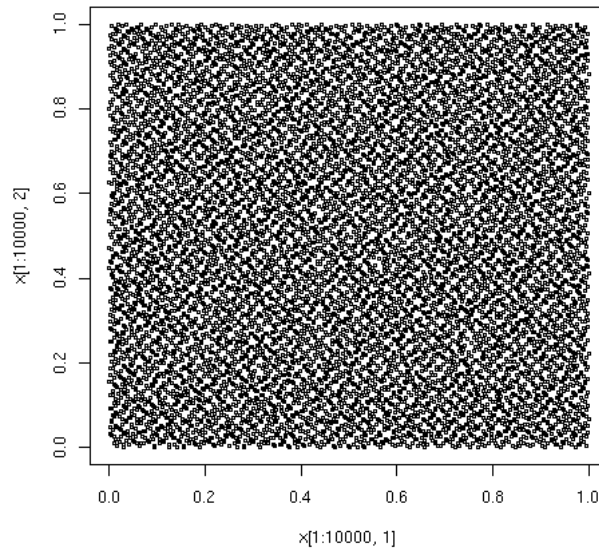
- Program and source: As usual, you must hand in everything needed to build and run your program, including all texture files and other resources.
- Gallery: Please put a few JPG pictures of the rendering results at least three. Please name the pictures ID-X.jpg (where X is a number).
- Read-me.txt:
 - Instructions on how to use your program (in case we want to use it when you're not around)
 - Descriptions of what your program does to meet all of the minimum requirements.
- Technical.txt:
 - The report could contain a description of this project, what you have learned from this project, description of the algorithm you implemented, implementation details, results (either good or bad), and what extensions you have implemented.

Advanced Technologies

- Sampling on area lights
Sampling is used to approximate an integral of a function as the average of the function evaluated at a set of points. Mathematically:

$$\int_0^1 f(u) du \approx \frac{1}{N} \sum_{i=1}^N f(x_i).$$

If $X_i = i/N$, the sampling is rectangular. If X_i is pseudo random or random, we call it as Monte Carlo sampling. If the sequence X_i has a low discrepancy, we term it as Quasi-Monte Carlo sampling. Loosely speaking, low discrepancy implies that a graphical representation of the sequence does not have regions of unequal sample density. For instance, consider the images below: the image to the left has low discrepancy.



1. Halton sampling is a Quasi Monte Carlo sampling technique that is deterministic. In 2D, it uses pairs of numbers generated from Halton sequences. These sequences are based on a prime number and can be constructed as follows: Pick a prime P and the number of desired samples N . Divide the interval $[0, 1]$ in this fashion: $1/p, 1/p^2, 2/p^2, \dots, p^2/p^2, 1/p^3 \dots$ till N unique fractions are created. To generate a sample in 2D, pick primes P, Q , generate the corresponding sequences and pair the numbers. It is recommended that the first 20 samples are discarded for higher primes due to a high correlation between those pairs.
2. Poisson Disk Sampling is a form of Poisson sampling where samples are guaranteed to be separated by a specified distance (radius). There are numerous techniques to generate Poisson Disk Samples efficiently such as [8] and [9]. However, for a low number of samples, we used the Dart Throwing technique and cache results. To do this, we generate points and discard those that do not meet the radius criterion. This process is continued till N points are reached. In our implementation, we create different sets of samples such that the same set is used for a specific bounce. We believe that this is similar to the approach taken in the original paper where each reflection uses samples based on a set of primes $(2j+2, 2j+3)$ where j is the reflection count.
3. Picking the right sampling is the key to getting impressive results using IR. Sampling is used in two areas in this project: to pick points on the light source, and to choose direction to shoot rays from the selected point. Each of these requires a mapping from samples on a unit square to those on triangles or on hemispheres. Fortunately, these are described in [Graphics Gems III](#) (relevant page available on Google books). These samples need to be weighted based on the area of triangle. Further, there are other technical considerations that affect scene independent implementations. For instance, increasing the number of samples per light source will result in a brighter scene unless the intensity of the original samples are weighted accordingly. Similarly, in open environments, the intensity of each light needs to be attenuated by the total number of created lights (and not estimated hits).

- [Lightcuts](#):



Lightcuts is a scalable framework for computing realistic illumination. It handles arbitrary geometry, non-diffuse materials, and illumination from a wide variety of sources including point lights, area lights, HDR environment maps, sun/sky models, and indirect illumination. At its core is a new algorithm for accurately approximating illumination from many point lights with a strongly sublinear cost. We show how a group of lights can be cheaply approximated while bounding the maximum approximation error. A binary light tree and perceptual metric are then used to adaptively partition the lights into groups to control the error vs. cost tradeoff.

We also introduce reconstruction cuts that exploit spatial coherence to accelerate the generation of anti-aliased images with complex illumination. Results are demonstrated for five complex scenes and show that lightcuts can accurately approximate hundreds of thousands of point lights using only a few hundred shadow rays. Reconstruction cuts can reduce the number of shadow rays to tens.

- Many light



Rerendering complex scenes with indirect illumination, high dynamic range environment lighting, and many direct light sources remains a challenging problem. Prior work has shown that all these effects can be approximated by many point lights. This paper presents a scalable solution to the many-light problem suitable for a GPU implementation. We view the problem as a large matrix of samplelight interactions; the ideal final image is the sum of the matrix columns. We propose an algorithm for approximating this sum by sampling entire rows and columns of the matrix on the GPU using shadow mapping. The key observation is that the inherent structure of the transfer matrix can be revealed by sampling just a small number of rows and columns. Our prototype implementation can compute the light transfer within a few seconds for scenes with indirect and environment illumination, area lights, complex geometry and arbitrary shaders. We believe this approach can be very useful for rapid previewing in applications like cinematic and architectural lighting design.

Reference

-
- [1] Keller, Alexander, "Instant Radiosity", Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, July, 1997
 [2] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila, "Incremental Instant Radiosity for Real-Time Indirect Illumination" in, "Proc. EGSR 2007", June 2007

Results



劉益銓, Incremental Instant Radiosity Implementation – 劉益銓

[Detail](#)



林德潔, Incremental Instant Radiosity Implementation -- 林德潔

[Detail](#)



呂仁傑, Incremental Instant Radiosity Implementation -- 呂仁傑

[Detail](#)



Hong-Wen Huang, Incremental Instant Radiosity Implementation -- 黃弘文

[Detail](#)

Copyright © 2019 NTUST CSIE Computer Graphics Lab. All right reserved.

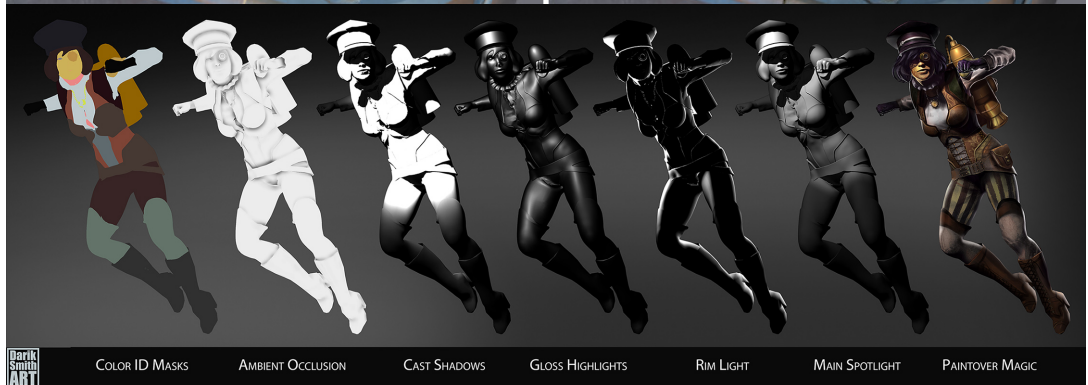
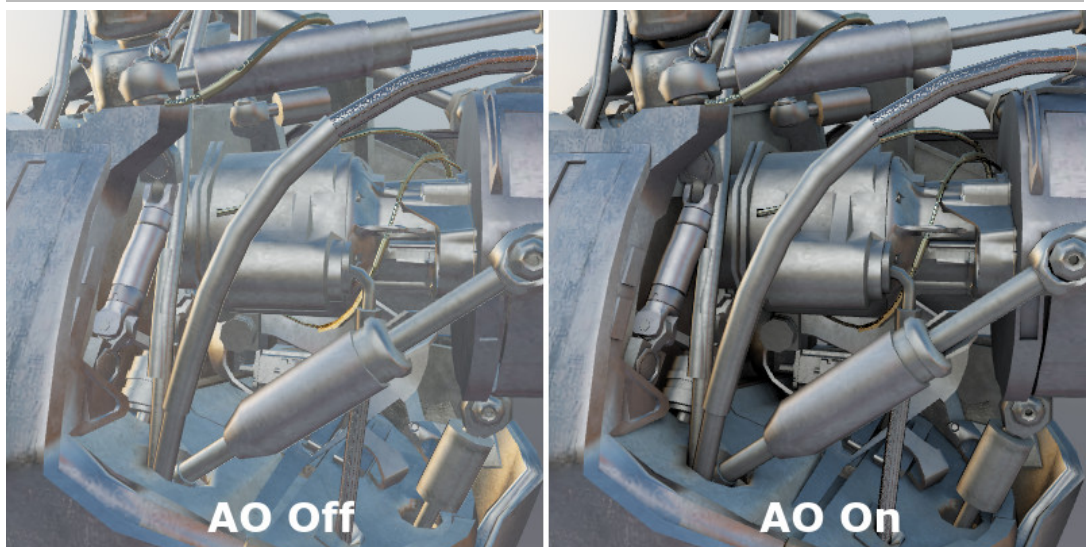


enu

- lome
- aculty
- tudents
- rojects
 - Research
 - Games
 - Others
- ourses

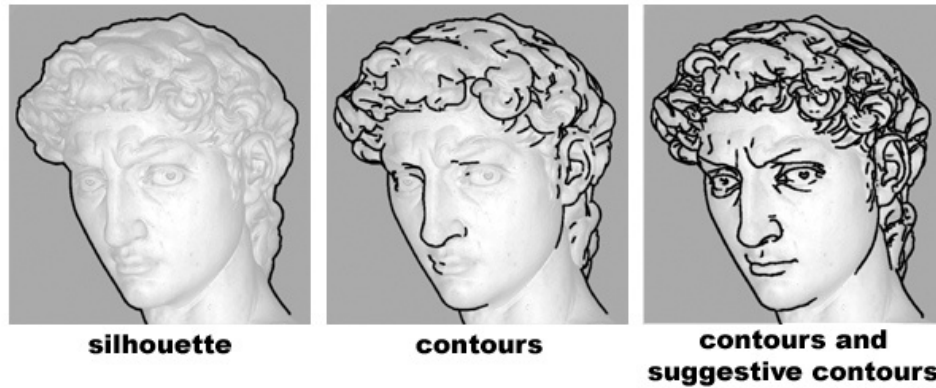
Project 4: Ambient Occlusion and Contour

Introduction



Ambient occlusion is a shading and rendering technique used to calculate how exposed each point in a scene is to ambient lighting. For example, the interior of a tube is typically more occluded (and hence darker) than the exposed outer surfaces, and the deeper you go inside the tube, the more occluded (and darker) the lighting becomes. Ambient occlusion can be seen as an accessibility value that is calculated for each surface point. In scenes with open sky this is done by estimating the amount of visible sky for each point, while in indoor environments only objects within a certain radius are taken into account and the walls are assumed to be the origin of the ambient light. The result is a diffuse, non-directional shading effect that casts no clear shadows but that darkens enclosed and sheltered areas and can affect the rendered image's overall tone. It is often used as a post-processing effect.

Unlike local methods such as Phong shading, ambient occlusion is a global method, meaning that the illumination at each point is a function of other geometry in the scene. However, it is a very crude approximation to full global illumination. The appearance achieved by ambient occlusion alone is similar to the way an object might appear on an overcast day.



When artists design imagery to portray a visual scene, they need not just render visual information veridically. They can select the visual cues to portray, and adapt the information each carries. Their results can depart dramatically from natural scenes, but can nevertheless convey visual information effectively, because viewers' perceptual inferences still work flexibly to arrive at a consistent understanding of the imagery.

We suggest that lines in line-drawings can convey information about shape in this indirect way, and work to develop tools for realizing such lines automatically in non-photorealistic rendering. In the figure above, the picture on the left renders silhouettes. The picture in the center renders occluding contours, and shows that contours, on their own, can be quite limited in the information they convey about shape. The picture on the right, however, includes additional lines we call *suggestive contours* that convey an object's shape consistently and precisely.

Pros and Cons

Screen-space Ambient Occlusion advantages:

- Advantages:
 - Independent from scene complexity.
 - No pre-processing, no memory allocation in RAM
 - Works with dynamic scenes
 - Works in the same way for every pixel
 - No CPU usage: executed completely on GPU
- Disadvantages:
 - Local and view-dependent (dependent on adjacent texel depths)
 - Hard to correctly smooth/blur out noise without interfering with depth discontinuities, such as object edges

Algorithm Overview

- Ambient Occlusion

Ambient occlusion is related to accessibility shading, which determines appearance based on how easy it is for a surface to be touched by various elements (e.g., dirt, light, etc.). It has been popularized in production animation due to its relative simplicity and efficiency. In the industry, ambient occlusion is often referred to as "sky light".

The ambient occlusion shading model has the nice property of offering a better perception of the 3D shape of the displayed objects. This was shown in a paper where the authors report the results of perceptual experiments showing that depth discrimination under diffuse uniform sky lighting is superior to that predicted by a direct lighting model.

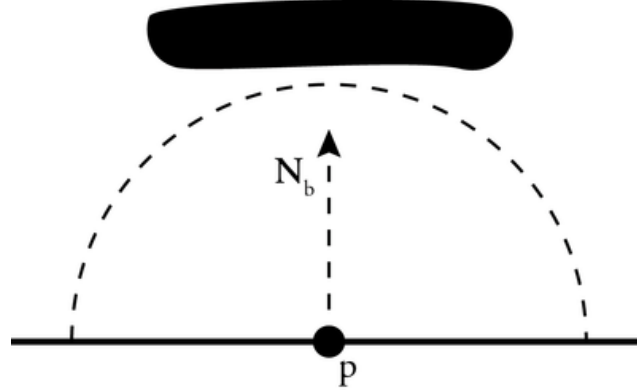
The occlusion $A_{\bar{p}}$ at a point \bar{p} on a surface with normal \hat{n} can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

$$A_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} V_{\bar{p}, \hat{\omega}}(\hat{n} \cdot \hat{\omega}) d\omega$$

where $V_{\bar{p}, \hat{\omega}}$ is the visibility function at \bar{p} , defined to be zero if \bar{p} is occluded in the direction $\hat{\omega}$ and one otherwise, and $d\omega$ is the infinitesimal solid angle step of the integration variable $\hat{\omega}$. A variety of techniques are used to approximate this integral in practice: perhaps the most straightforward way is to use the [Monte Carlo method](#) by casting rays from the point \bar{p} and testing for intersection with other scene geometry (i.e., [ray casting](#)). Another approach (more suited to hardware acceleration) is to render the view from \bar{p} by rasterizing black geometry against a white background and taking the (cosine-weighted) average of rasterized fragments. This approach is an example of a "gathering" or "inside-out" approach, whereas

other algorithms (such as depth-map ambient occlusion) employ "scattering" or "outside-in" techniques.

In addition to the ambient occlusion value, a "bent normal" vector \hat{n}_b is often generated, which points in the average direction of unoccluded samples. The bent normal can be used to look up incident radiance from an environment map to approximate image-based lighting. However, there are some situations in which the direction of the bent normal is a misrepresentation of the dominant direction of illumination, e.g.,



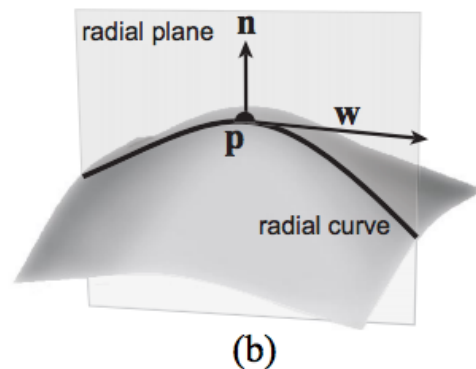
In this example the bent normal N_b has an unfortunate direction, since it is pointing at an occluded surface.

In this example, light may reach the point p only from the left or right sides, but the bent normal points to the average of those two sources, which is, unfortunately, directly toward the obstruction.

- Silhouette and Suggestive contour:

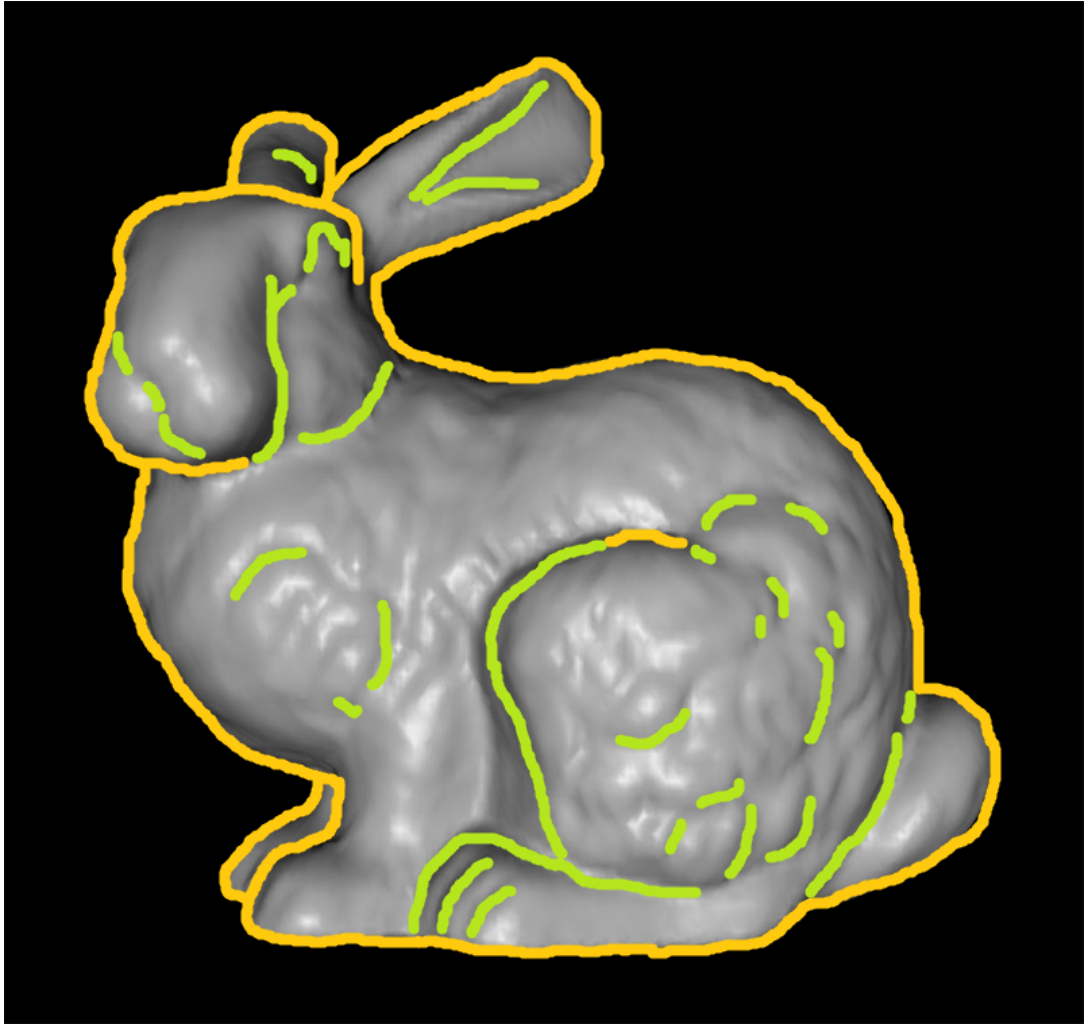


A non-photorealistic rendering system to convey shape using suggestive contours and highlights. Suggestive contours are lines that will most likely be contours if viewed at a different angle.



Contours are edges that join a polygon facing the viewer with one facing away. Drawing the contours of a model is pretty simple, but gives you little more than a silhouette. Suggestive contours are "almost

contours" or edges that would become contours in relatively nearby viewpoints. They give much more meaningful information about a model's shape. In the bunny on the right, contours are drawn in yellow, and suggestive contours in green. Another example of contours versus suggestive contours is below.





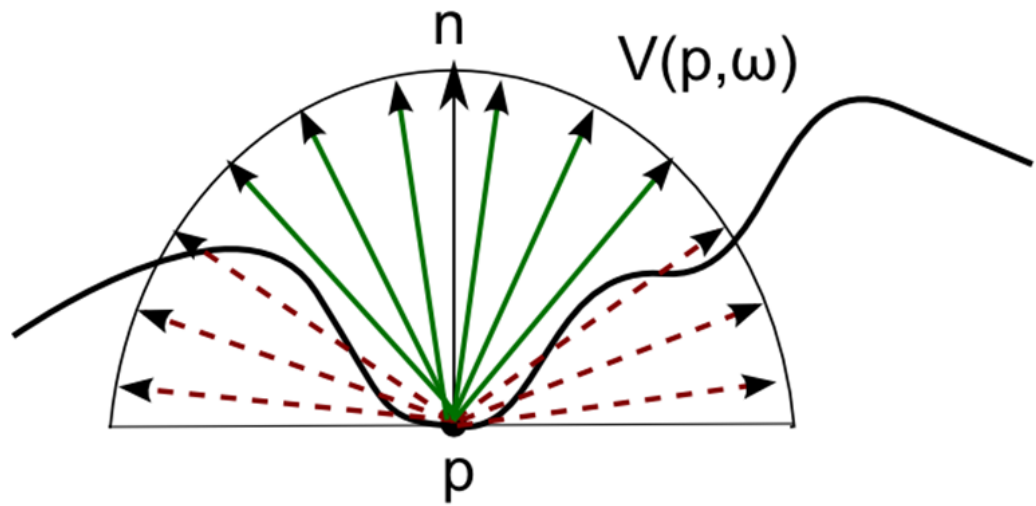
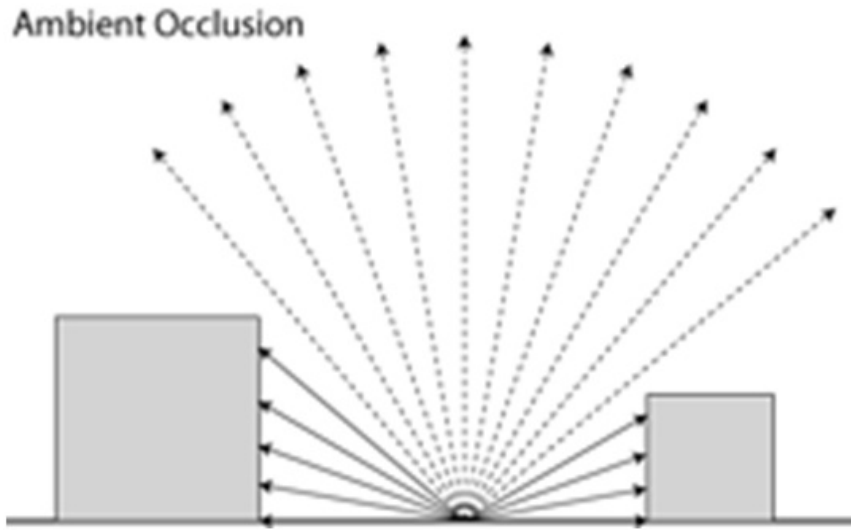
Contours on the left, and suggestive contours on the right.

The Basic Task

- Ambient Occlusion

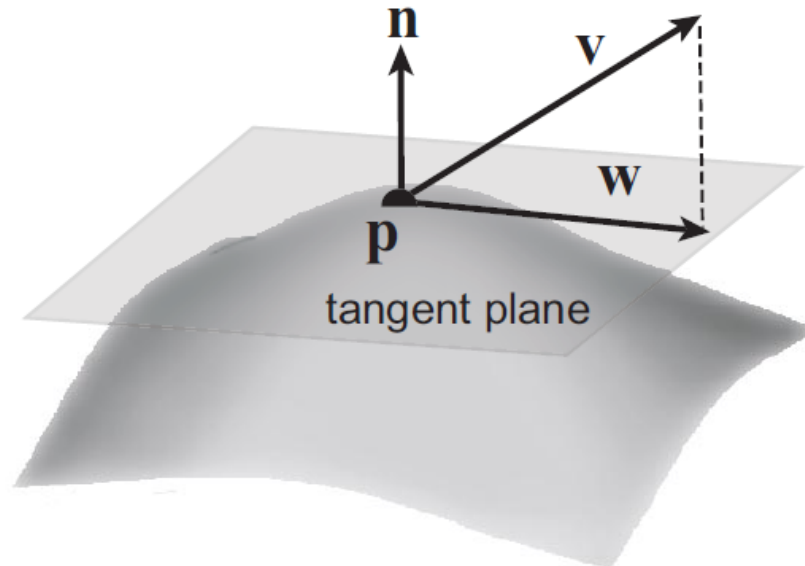
```

For each hit point {
  Shoot out N rays with radius R and for each ray {
    Check the distance to the hit point.
    If radius is smaller than R
    Count 1
  }
  Determine the ambient occlusion based on the ray counting
}
    
```



$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega) \mathbf{n} \cdot \omega d\omega,$$

- Silhouette and Suggestive contours: Implementation provides a bit more detail about how contours and suggestive contours are calculated, and how the lines are drawn.
- Contours and Suggestive Contours



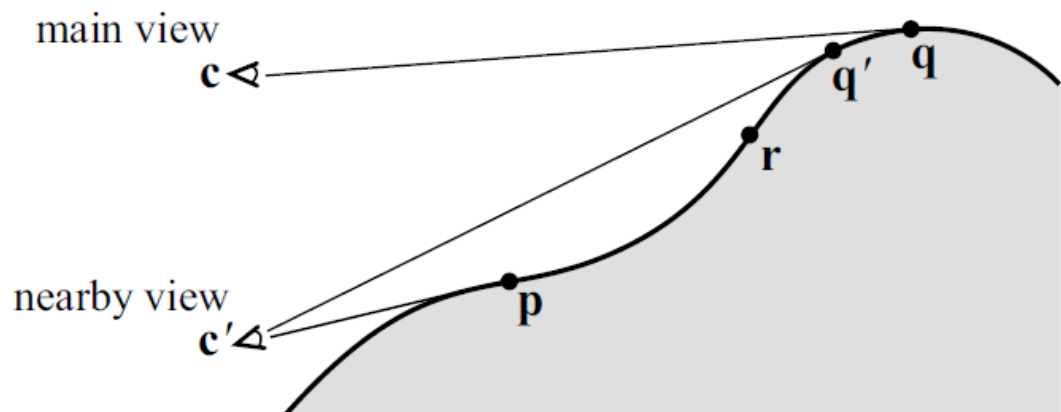
To draw the contour

lines, the first step is to calculate $n \cdot v$ for every point on the mesh. The vector n is the normal vector at the point, and v is the view vector (see the figure at the above).

Contours are the places where $n \cdot v = 0$. Suggestive contours, mathematically, are the set of points on the surface at which:

1. The radial curvature K_r at the point is 0 (meaning there is a point of inflection along the curve)
2. $D_w K_r > 0$ (meaning that the curve switches from being convex - like a mountain - to concave - like a valley)

In the picture below (from [2]), the point p is part of the suggestive contour.



Note that you also need to cull the lines for contours and suggestive contours on the back of the mesh.

Require Functionalities

1. Parse in the scene, material, light and other informations.
2. Fast local rendering with OpenGL.
3. Shoot out ambient ray from each hit points to collect the ambient occlusion with Monte Carlo methods to determine the ambient occlusion. (15%)
4. Rendering the scene with ambient occlusion (5%)
5. Analyze the each object mesh to determine their silhouette. (5%)
6. Determine the suggestive contours (20%)
7. Rendering with line drawing using silhouettes and suggestive contours (5%)
8. Implement Screen-space ambient occlusion with GPU. (15%)
9. Implement GPU-based suggestive contours. (15%)

What to Hand in

All your hand-in must be put in a directory with your student ID and the following is the list of hand-in files under the directory.

- Program and source: As usual, you must hand in everything needed to build and run your program, including all texture files and other resources.

- Gallery: Please put a few JPG pictures of the rendering results at least three. Please name the pictures ID-X.jpg (where X is a number).
- Read-me.txt:
 - Instructions on how to use your program (in case we want to use it when you're not around)
 - Descriptions of what your program does to meet all of the minimum requirements.
- Technical.txt:
 - The report could contain a description of this project, what you have learned from this project, description of the algorithm you implemented, implementation details, results (either good or bad), and what extensions you have implemented.

Advance technologies

- Ambient occlusion
 - [Ambient Occlusion Volumes](#)
This paper introduces a new approximation algorithm for the near-field ambient occlusion problem. It combines known pieces in a new way to achieve substantially improved quality over fast methods and substantially improved performance compared to accurate methods. Intuitively, it computes the analog of a shadow volume for ambient light around each polygon, and then applies a tunable occlusion function within the region it encloses. The algorithm operates on dynamic triangle meshes and produces output that is comparable to ray traced occlusion for many scenes. The algorithm's performance on modern GPUs is largely independent of geometric complexity and is dominated by fill rate, as is the case with most deferred shading algorithms.
- Suggestive contour extension
 1. Pen and Ink Shading: Often artists do pen and ink shading by drawing cross-hatching or parallel lines to indicate a shadowed region. I tried to imitate this.
 2. Varied Line Thickness: I tried varying the line thickness depending upon how "lit" the line was (so the more in shadow, the thicker the line).
 3. Varied Color: I tried to vary the shade of the line depending upon how lit the line was
 4. Implement with Chinese ink painting.

References

1. Miller, Gavin (1994). "Efficient algorithms for local and global accessibility shading". Proceedings of the 21st annual conference on Computer graphics and interactive techniques. pp. 319–326.
2. Langer, M.S.; H. H. Buelthoff (2000). "Depth discrimination from shading under diffuse lighting". Perception. 29 (6): 649–660. doi:10.1068/p3060. PMID 11040949.
3. Oscar 2010: Scientific and Technical Awards, Alt Film Guide, Jan 7, 2010
4. Suggestive Contours for Conveying
Shape: http://gfx.cs.princeton.edu/pubs/DeCarlo_2003_SCF/DeCarlo2003.pdf
5. Highlight Lines for Conveying
Shape: http://gfx.cs.princeton.edu/gfx/pubs/DeCarlo_2007_HLF/highlights_npar07.pdf
6. Dan Maljovec: <http://www.cs.utah.edu/~maljovec/CS6610/>
7. Suggestive Contours by Alyssa
Daw: <http://users.csc.calpoly.edu/~zwood/teaching/csc572/final10/acdaw/>

Results

林德潔, and 劉益銓, Ambient Occlusion and Contour - 林德潔_劉益銓



Detail

呂仁傑, Ambient Occlusion and Contour -- 呂仁傑



Detail

Chia-Hsing Chiu, Ambient Occlusion and Contour -- 邱嘉興



Detail



Menu

- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 5: Motion Path Editing

Overview

The purpose of this project is to give you some experience dealing with motion capture data, to experience the issues in using file formats for motion data, to gain intuitions about how motion editing techniques work, and to work through the details of an animation methodology from a research paper.

The Basic Idea:

You must write a program that reads BVH files (a standard skeletal animation data file format) and displays them in an interactive viewer. You must implement a variant of path editing to allow the user to alter the motion. You must implement some kind of motion blending and the ability to concatenate motions (play motions one after another).

Ground Rules

Your program must run on the Windows machines with Unity, Unreal, Ogre3D, and QT OpenGL. You may use utilities and libraries that you find, subject to approval. Any "borrowed" code must be clearly documented. Copying someone's assignment from a previous year is not OK.

Reading Files

Your program must read "Biovision hierarchy files." In the webpage [motion data](#), you will find lots of BVH files to experiment with. You will see all sorts of different skeletal configurations, joint types, and other variations of the file formats. In the ideal world, you would be able to read any BVH file we threw at your program. More realistically, we would like your program to understand some subset of them. The larger the subset, the better (and this will be rewarded in grades). Some ground ground rules:

1. Your program should not crash on a file that it cannot understand.
2. Your program must read at least 5 of the files in the Motion directory. You can pick the 5.
3. You must, in your documentation, describe the limits of the reader. It is much better to say "my reader only supports ZXY euler angles for joints" than to have a reader that mysteriously doesn't work some times.
4. Some reasonable simplifying assumptions you might make include: supporting only certain types of Euler angles, or only supporting a fixed topology. (the latter is a severe limitation, and I do not recommend it).

Displaying Motion

You must provide an interactive viewer for displaying the motions that you read in. Your viewer should display things, and provide the ability to play the motion at "frame rate" as well as the ability to "scrub" (interactively move through frames). You must provide some interactive camera controls so that the user can control the view.

The nicer that you draw things, the better. Drawing lines between the joints is the easiest, but drawing some "bones" (like ellipsoids) looks a lot better. Skinning looks the best, but that's a really advanced feature.

1. You should do some things to help make the motion easier to see - for example, drawing "traces" that sweep out the paths of the end-effectors, or strobes (drawing several frames simultaneously). Try being creative in given tools to help the user visualize the motion.

2. Drawing a groundplane and shadows are an easy way to make things look a lot better.
3. Your program must be able to place the camera in a position where it can see the whole motion (this requires you to figure out the spatial extents of the motion). The better your program does at this, the better.
4. A nice advanced feature to add is a tracking camera that follows the character as it moves. This takes a little thought to do well - it can't be too bouncy, or spin too fast, ...
5. You must put in a timer to control framerate. Your program must offer the option of displaying animation at 30fps. You might want to allow other options (like just blasting the animation as fast as possible, or allowing for slow motion viewing).
6. Your program should also support showing 2 (or more) motions at once. This will be useful for when two characters interact, as well as for comparing motions (which you will need to do).

Forward Kinematics

Your program must compute the forward kinematics. That is, you must be able to compute where each joint goes, and have some way to show that you can do this. For example, you might want to draw a "trace" line showing where the point goes.

Path Editing

You must provide the user with a way to edit the path of the motion (e.g. if the character was walking in a straight line, you can bend it and have it walk along a curve). The paper on path editing is here. Note: this is not a paper we will be discussing in class. Part of the exercise is for you to learn how to read a technical paper, figure out how it works, and try it out. While I am biased (I did write the paper), I think that this one isn't too hard.

Path editing has a lot of variants. Start with the most basic, and then add features. The most basic version (ignoring orientation) should be trivial. I expect that most people will be able to get the orientation control. Arc-length parameterization (you'll understand what I'm talking about after you read the paper) is clearly an advanced thing. Putting in some kind of IK solver to cure foot skate is a very advanced feature (but there are some simple ways to do it that might make it not so bad). There are many other possible extensions to path editing to try. For example, you might determine when the character is in the air, and make sure the path doesn't bend (ideally, you could stretch the path in such a case). You might mark parts of the motion that are not allowed to be path edited (for example, when a character stops to pick up an object).

Rigid Transformations, Concatenating and Blending

You need to be able to apply a rigid transformation to a motion before displaying it. This will be important for blending and concatenating. You should have some interface for manually editing motions. You need to be able to play a sequence of motions one after another. This involves applying the transformation such that the end of one motion is the beginning of the next. We will provide you with examples of pieces of motion that simply "snap together" - you will be able to make loops and whatnot using them by playing one after another. You might want to be able to set up your interface so that you can load in a bunch of motions and play sequences of them.

You need to be able to blend 2 motions together (with varying blend weights). For example, to make a transition between one motion and another. This means that you must be able to rigidly transform motions (so that they line up well enough) and time shift things. If you want to do better (implement time warps or something that's a bonus).

Note: the blending can be completely manual. However, you must be able to position the two motions relative to one another, and set the time shift.

Requirements

You must turn in all files required so that we can run your program, documentation on how to use your program (including a description of limitations), and a description of your program's features.

The basic project would include:

1. Reads many BVH files, with some restrictions
2. Be able to read in 2 BVH files simultaneously and show them together.
3. Displays ellipsoids or other rigid shapes for bones
4. Places the camera automatically, and gives a camera user interface
5. Implements basic path editing (with orientations handled correctly)
6. Is able to position motions (under user control) and concatenate and blend them

It is better to have a project that does all of the basic features and works well, than to have a project that has some fancy, advanced feature, but fails at the basics.

An advanced project might add:

1. Read almost all BVH files
2. Display nice character geometry
3. Have a good camera UI that includes tracking
4. Implements advanced path editing (arc-length parameterizations)

Above and beyond the call of duty projects might:

1. Does better alignment methods (registration curves, time warps), or automatic methods to find alignments
2. Has some methods for choosing which motions to concatenate (interactive control, random walks, ...)
3. Implement Inverse Kinematics to clean up footskate
4. Do smooth skinning
5. Provide other motion editing features

How will we grade it?

You will have to provide a web page documenting the project (details to follow). You will also have to give a in-class demo of your project.

Reference

- [1] Michael Gleicher, "Motion Path Editing", In Proceeding of 2001 I3D.
- [2] BVH Motion Data Sets [bvh_sample_files.zip](#)
- [3] BVH Information Slides [NTUST-CSCG2011S-P2-Steps.ppt](#)
- [4] BVH Introduction. [BiovisionHierarchy.pptx](#)

Copyright © 2019 NTUST CSIE Computer Graphics Lab. All right reserved.



enu

home

faculty

students

projects

• Research

• Games

• Others

courses

Project 6: Particle Simulation with Physics Engine

Overview

The goal of this project is to give you a chance to explore the issues in doing physical simulation for animation. You must implement the basic requirement which builds a particle simulation system in 3D space like a cloth (each particle is a point mass that can have forces pushing it). For this assignment, trying different simulation methods is more important than making fancy images. The project does have various stages, that you should progress through. However, you should look ahead to make sure that your design decisions in an early phase don't preclude what you will need to do later. The early phase of the project will have you build a particle simulator where you use various forces on the particles to move them around. You should try to make your system interactive and fast so you can try lots of things. The later phase of the project will have you experiment with making a complex mass-spring simulation such as cloth. You will be able to make a springy string using the early part of the project. But if you want to make those springs connecting the particles of the string together stiff, you will need to explore different solution methods: explicit integrators (such as predictor-corrector), implicit integration (ala. Baraff&Witkin's cloth paper), and semi-implicit integration (Bridson Fedkiw).

- Note: You need to implement this particle simulation system with Unity, Unreal, OGRE3D, and OpenGL.

Phase 1: build an interactive particle simulator.

The system must be able to simulate a number of particles - each particle should be able to have its mass set to a different value. You should be able to add particles interactively. In addition, the system must have interface to show the progress of the simulation.

Requirement for this phase:

Your code must implement the following features:

- Simulate a number of particles - each particle should be able to have its mass set to a different value. You should be able to add particles interactively.
- Show the progress of the simulation - this is important for debugging.
- Save and read in a file that describes the initial configuration of the particles, as well as any forces to be applied to them.
- Save the results of a simulation and replay it. While your simulation does not have to be real time, your playback should be able to play at the correct frame rate for your simulation.
 - Note: you may want to have an adaptive step-size integrator at some point, so make sure your playback understands the simulator timing.
- A generalized force structure: This is described in the slides. (If you're using the skeleton code, you should replace `delete_this_dummy_spring` with a `std::vector` of forces.) You must implement two subclass forces:
 - Constant force (constant in a particular direction): such as gravity force acting like gravity.
 - Damping force: Velocity-dependent damping force
 - Spring force:

1. A spring (of a setable distance) between two particles: (e.g. a force proportional to the relative velocity of particles) repulsion between particles (if the distance between particles is ever less than X , they push each other away)
 2. A spring between a particle and a specific location (or, allow some particles to be un-movable nails and then you just need springs between particles)
 3. A spring that pushes particles upward from the floor (penalty collisions with the floor), or outward from the walls.
- A generalized constraint structure: This is also described in the slides. (If you're using the skeleton code, you should replace `delete_this_dummy_rod` and `delete_this_dummy_wire` with a `std::vector` of forces.) You must implement at least the following two subclasses:
 - `RodConstraint`. Constrains two particles to be a fixed distance apart. (Rendering code included in the skeleton.)
 - $C(x_1, y_1, x_2, y_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 - r^2$
 - `CircularWireConstraint`. Constrains a particle to be a fixed distance from some point:
 - $C(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2$
 - Mouse interaction. When the user clicks and drags the mouse, a spring force should be applied between the mouse position and the given particle to make your system interactive.
 - Several Numerical Integration Schemes (Simulators). The integration scheme should be selectable at runtime with keystrokes or some other interaction paradigm. You will find this easiest if you implement a pluggable integration architecture as described in the slides. The minimum integration schemes are:
 - Euler
 - Runge-Kutta 2 and
 - Runge-Kutta 4 and
 - The user interface should be able to draw all of the connections/forces. Especially the spring connections.

Doable demo in this phase

1. Make a "soft" object by taking some particles and connecting them with a lattice of springs. the simplest is a triangle. throw the object around and watch it bounce off the walls and jiggle.
 2. Make a fountain of particles - use repulsion to make a "fluid". the stream of the fountain should wind up in a puddle of particles at the bottom. (use damping to make sure that things don't explode)
 3. Make a swinging rope or chain. connect a line of particles to their neighbors, and nail the top on in place. (use a little damping to prevent explosions)
- Note: At this point, everything should be either very springy or very damped, since you are still using simple integration methods.

Phase 2: Stiff Spring

In this phase, the first step is to improve the integrator by

- Implement an implicit euler's method (as in Baraff and Witkin). Because the system is so small, you need not worry about their efficiency tricks for large sparse linear systems. Once you have an implicit solver, a semi-implicit solver should be easy (use explicit solution of the springs and gravity, and implicit solution of damping - as in Bridson&Fedkiw).

Doing the following experiments

- Make a chain of 10-20 particles, connected with springs and a little bit of damping. Nail the top one in place, and let this "rope" swing. Make a graph of the total length of the chain. Notice how it stretches and compresses. The ability to measure this stretching will be useful as we explore different integrators.
- Try to raise the stiffness of the springs to prevent stretchiness. At some point, the equations will become too stiff and the string will explode.
- Experiment with the tradeoff between stiffness and timestep.

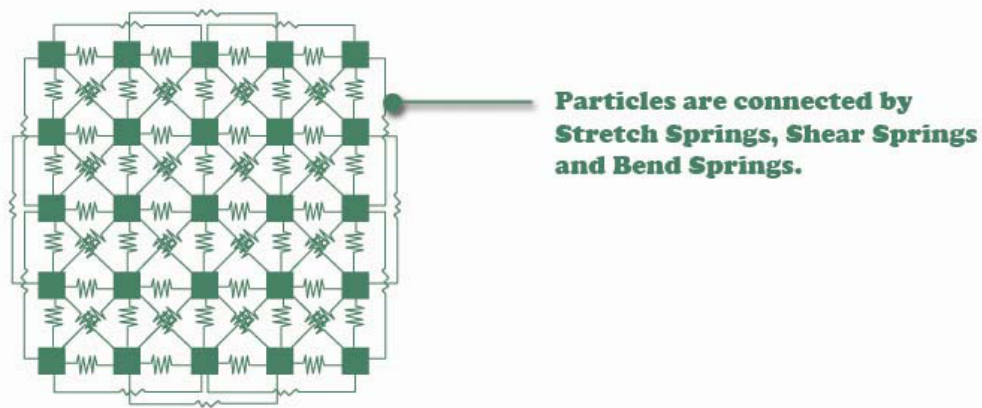
- You now have the ability to experiment with the various tradeoffs in creating a stiff piece of string. It should be the case that with better integrators (like implicit), you can take larger time steps, but each time step takes longer to compute. An implicit solver might allow you to take time steps that at 100 times bigger than an explicit solver, but it might take 200 times longer to compute (in which case, its probably better to do 100 explicit steps).
- For this part of the assignment, you need to report on what you find. What are the tradeoffs? How much better are different solvers? Is implicit necessarily better than explicit? Is one RK4 step better than 4 Euler steps (which should be approximately the same amount of computation).

Phase 3: Create a cloth simulation

Mass-Spring Model

The Mass-Spring Model is a very basic method to simulate cloth. Since it is relatively easy to implement and could achieve good results, implementing this model could be a very good exercise before exploring some more advanced cloth simulation methods.

In this model, cloth is simulated by a grid of particles which are interconnected with spring-dampers. Each spring-damper connects two particles and generates a force based on the particles' positions and velocities. The structure of this model could be illustrated using the image below:



Each particle is also influenced by gravity. With these basic forces, we can form a cloth system.

Particle

The starter code already provides you with a particle class. Each particle has several properties at a specific time, including its position, velocity, mass and a force accumulator which stores all the forces influence the particle at that time. From these properties, we can calculate how particles would move at next time step.

Computing Forces

1. Gravity:

Computing gravity is simple, we can use this equation:

$$\mathbf{f}_{\text{gravity}} = m\mathbf{g}$$

2. Spring-Dampers:

A spring-damper connects two particles, it has three constants defining its behavior:

Spring constant K_s , Damping factor K_d and Rest length l_0

The equation below gives you the forces the spring-damper exerts on the two particles:

$$\mathbf{f}_1 = - \left[k_s (|\mathbf{p}_1 - \mathbf{p}_2| - l_0) + k_d \left(\frac{(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{p}_1 - \mathbf{p}_2)}{|\mathbf{p}_1 - \mathbf{p}_2|} \right) \right] \cdot \frac{(\mathbf{p}_1 - \mathbf{p}_2)}{|\mathbf{p}_1 - \mathbf{p}_2|}$$

$$\mathbf{f}_1 = -\mathbf{f}_2$$

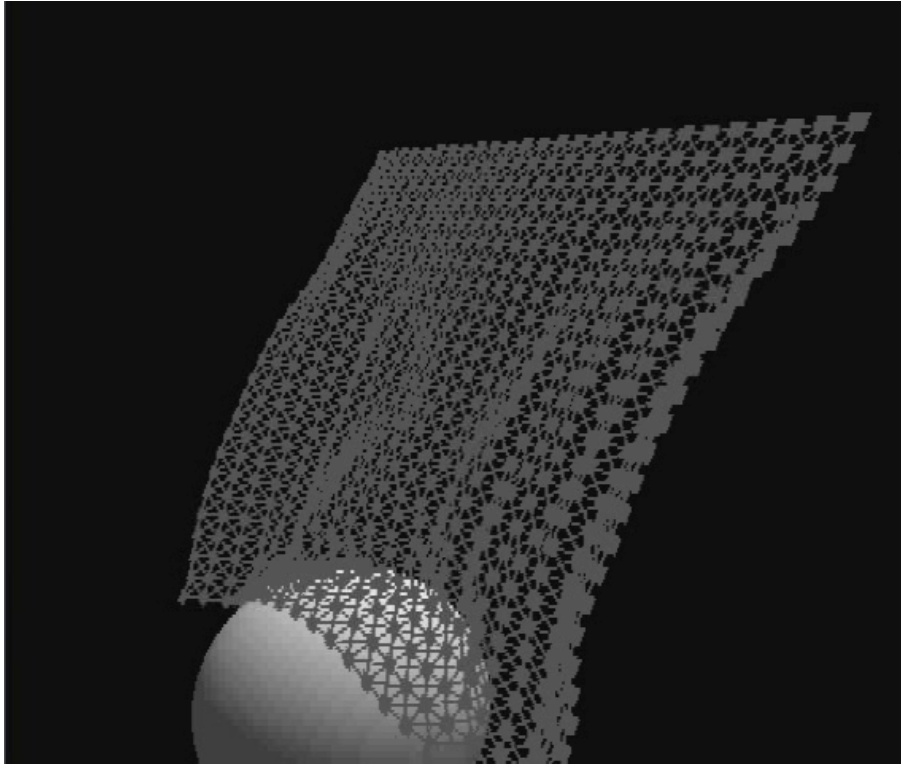
Cloth Simulation

The algorithm of simulating cloth could be summarized as

1. Compute forces for each particle
2. Integrate motion: apply Forward Euler Integration for each particle
3. Repeat

Reference Image

(Note: for the basic requirements, you don't need to implement collision.)



- There should be existing cloth simulation existing in your physical engine and try to use them to create a similar simulation to the one you created in phase 2 and do the comparison.

Extra Credit

- Better integrator (+ 5 Each)
 1. Verlet Integrator. See [here](#).
 2. Leapfrog Integrator. Evaluates position and velocity at different times. See [here](#) for more details.
 1. Symplectic Integrator. As described in class. Compute the positions explicitly and velocities implicitly. (No need for a solver.)
- Collision (+ 10 Each)
 1. Collisions with the Walls. Particles should bounce off the walls and floor.
 2. Collisions with other Particles. Particles bounce off each other.
- Angular Springs (+15). Pulls a triplet of particles so that their subtending angle approaches some rest angle.
 - Angular Constraints (+20). Like angular springs, but the angle is actually constrained.
- 3D Cloth with collisions.(+30)
- Hair with collisions.(+30) How can this be implemented? What about collisions?

What you will turn in

1. You will prepare a web page describing you assignment, complete with pictures of what your program looks like and a description of its features and user interface. Be sure to detail all of the integrators that work, and all the types of forces that the user can create. You should also document your file format for describing simulation initial conditions.
2. You must also write a report (either in html or a pdf) that is linked to this page describing your experiments and results for phase 2. Describe how to maximize stiffness, minimize damping, and maximize performance. What are the tradeoffs? Be as detailed as possible.
3. Wherever possible, provide quantitative details. And, be sure to discuss how what you find compares with what you would expect to find.
4. Also, consider the computational costs. At what point would it be useful to switch to a sparse linear solver? how would the size of the system that you were simulating effect the choices in integrator?



Menu

- Home
- Faculty
- Students
- Projects
 - Research
 - Games
 - Others
- Courses

Project 7: Chain Reaction

Overview

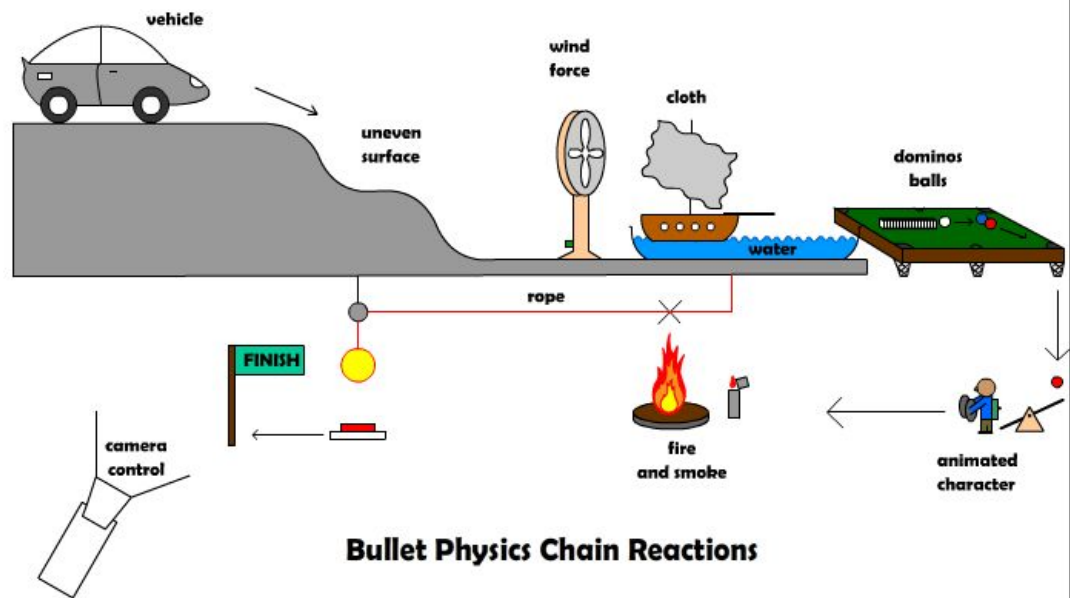
In this project, you will make the roller coaster applications more interesting. You will add an interactively controlled animation into the roller coaster. The purpose of this project is to force you to use the physical simulation engine which is an important tool for the game design. The animation is to demonstrate the a chain reaction simulation. The things must be done with a physical engine in QT & OpenGL. We recommend you to use bullet.

Components in your animation.

- Vehicle moving on uneven surface:
 1. Vehicle must be export from Maya, 3DMax or Blender and then be loaded in by the OpenGL function.
 2. You should also have some UI to control the speed of the car.
 3. Uneven surface can be a terrain using the height field.
 4. You should provide some UI to control the roughness and the slope of the ground
- Cloth and wind
 1. Cloth simulation can be easily done by the physical engine and thus in your scene, you must get the cloth into the reaction chain.
 2. The wind is the force to act on the cloth and the speed of the wind must be controlled by a slider. In addition, the speed of the wind will also define the force on the cloth.
- Water (not real simulation)
 1. Water is important for reality. Water simulation is slow but you can have different ways to hack the effect such as using the combination of sine wave. More examples can be found in Bullet and NVidia web site.
 2. Refraction effect on the water.
- Some objects like dominoes and pool balls
 1. Dominoes can act a complex reaction among objects which one cause another to react and this is interested physical phenomenon.
 2. The same description can be say to pool balls.
- Animated character: skeleton characters are important in the game. It can represent the hero or the enemy. Please use a tool to import the animation in and put an animation on the character.
- Fire and smoke:
 1. Fluid is important for reality and normally a physical engine will provide the proper tool to do it.
 2. The size of the fire and the heat generated by the fire should be able to control by a slider
- Camera control to follow the active animation: the smooth game camera motion is important to generate a nice game and thus you need to have a follow camera to follow the events. In addition, the camera should be moved smoothly.
- Sound according to the events: sound is an important element. You should put the sound for each collision and the events happen.
- Sky
- Day and night: day and night should have different effects on the result.

Example scenarios

Here we will provide an example scenario but we hope that you can make something more interesting and intrigued. The scenario setting is illustrated in the figure.



1. When the user push the button, the car is initiated down the slope and the UI similar to accelerate pedal should be able to control the speed.
2. The car is driven down the hill which is an uneven surface.
3. The car will hit the start button of a fan and the fan start to rotate to generate the wind and there should be a UI to control the speed of the wind.
4. The wind will blow the cloth to make the boat move across the water and the speed of the wind will change the speed of the boat.
5. At the same time, when boat move across the water, it should show the wave as the ripple. At the same time the water should reflect the sky and show the bottom of the water with refraction.
6. The boat will cross the water and then initiate the dominos and then hit the balls to start the robot.
7. The robot will walk across with some interesting animation to hit the start of the fire.
8. The fire is up and will burn down the rope and the size and heat of the fire should be controlled by some UI.
9. The fire will burn down the rope and the weight will drop and stop the animation and then there should be some final scene.

Check Points

1. Check point 1: integrate bullet: Demo basic object collisions (cubes, spheres, plane).
2. Check point 2: set up the scene layout: Model the terrain, import different models (vehicle, table, boat, etc.); Show that everything works with the physics engine.
3. Check point 3: have the vehicle simulation, camera control, and UI for setting parameters.
4. Check point 4: demonstrate all physics simulations (cloth, water, fire, smoke, etc.).
5. Final demo.

教學與服務記錄

國立台灣科技大學 資訊工程學系

賴祐吉 副教授

2019 年 12 月

教學

A. 開授課程 (99 學年度至 103 學年度)

● 研究所

- ◇ 3D電腦遊戲(一)，3D Computer Game I (程式實作代替考試)
(104上(英)：4.9、105上(英)：3.68、106上：3.93、107下(英)(P)：4.33)
- ◇ 3D電腦遊戲(二)，3D Computer Game II (程式實作代替考試)
(104下(英)：4.40、105下(英)：4.20、106下：4.00)
- ◇ 計算機圖學，Computer Graphics (程式實作代替考試)
(105上：3.60、106上：4.43、107上：4.46)
- ◇ 數位網格處理，Digital Mesh Processing(程式實作代替考試)
(104上：5.00)
- ◇ 資工研究與實務校外實習(一)，Research Internship in Computer Science and Information Engineering (I)
(107上)
- ◇ 資工研究與實務校外實習(二)，Research Internship in Computer Science and Information Engineering (II)
(107下)
- ◇ 資工研究與實務校外實習(四)，Research Internship in Computer Science and Information Engineering (IV)
(106下：4.00)

※括弧內，黑體數字，為教學評量成績，(英)，為英文授課，(P)為PBL課程。

● 大學部

- ◇ 電腦圖學導論，Introduction to Computer Graphics，(程式實作代替考試)
(104下：4.38、105下：4.46、106上：4.02、107上：3.89)
- ◇ 電腦圖學導論實習，Introduction to Computer Graphics Lab，(程式實作代替考試)
(104下：4.44、105下：4.27、106上：3.88、107上：3.94)
- ◇ 物件導向程式設計，Object-oriented Programming，(期中期末，外加程式實作)

(105下：3.55、105下：3.74、107下：3.94)

- ◇ 物件導向程式設計實習，Lab for Object-oriented Programming，(程式實作)

(105下：3.60、105下：3.81、107下：4.04)

- ◇ 遊戲企劃與設計原理實習，Lab for Computer Game Design and Development，(程式實作取代考試)

(106上：3.74)

- ◇ 電腦遊戲賞析，Game Survey，(賞析遊戲報告及影像剪輯)

(106上(P)：4.12)

- ◇ 電子電路，Electronics and Electrical Circuit，(期中期末，外加程式實作)

(104下：3.94)

- ◇ 手機遊戲設計，Mobile Game Design (程式實作代替考試)

(104上：4.08、105上：3.70)

- ◇ 遊戲程式設計，Computer Game Programming (程式實作代替考)

(106下(P)：3.79、107下：4.06)

- ◇ 專業成長校外實習，Summer Intern for Professional Training

(104上、104下)

- ◇ 資工實務暑期校外實習，Summer Practical Training for Computer Science and Information Engineering

(104暑)

※括弧內，黑體數字，為教學評量成績，(P)為PBL課程。

- 協助執行教育部之「ILab」計畫，幫忙建構實驗室及考試之試行。
- 協助執行教育部之「高階人材」計畫，協助完成KPI。
- 教育部「產業學院」試辦計畫，電資學院開設短期契合式產學專班學程，合作公司，鈦象電子股份有限公司，互動遊戲設計學程。
- 與姚智原、戴文凱和花凱龍教授合作，於108年度成立AI跨域應用產業碩士專班。
- 教育部智慧製造電子應用聯盟中心智慧整合感控系統工業應用分項執行機械手臂設計環境之擬真生成、探測深度產生和虛擬實境上之展示。
- 課程所邀請之業師

業師	任職	課程
吳育光	鈦象電子股份有限公司/研發部長	手機遊戲設計(104上、105上) 計算機圖學(105上)
曾冠諦	鈦象電子股份有限公司/資深主任	手機遊戲設計(104上、105上) 計算機圖學(105上) 電腦圖學導論實習(106上) 計算機圖學(106上)

		3D電腦遊戲 (二)(106下) 遊戲程式設計(106下)
王峻偉	鈦象電子股份有限公司/高級軟體工程師 天紫科技有限公司/資深軟體工程師	3D電腦遊戲 (一)(105上、107下、108上) 3D電腦遊戲 (二)(105下) 電腦圖學導論(107上、108上) 電腦圖學導論實習(107上、108上) 計算機圖學(107上) 物件導向程式設計(107下) 物件導向程式設計實習(107下) 遊戲程式設計(107下)
林傳健	7QUARK/CTO & Co-Founder	電腦圖學導論(105下) 電腦遊戲賞析(106上) 遊戲企劃與設計原理實習(106上) 3D電腦遊戲 (一)(106上) 3D電腦遊戲 (二)(106下) 遊戲程式設計(106下)
曾柏達	神研科技/技術經理	物件導向程式設計(105下、106下) 物件導向程式設計實習(106下)
陳國璋	台灣盈米科技股份有限公司/專案經理	物件導向程式設計(105下) 3D電腦遊戲 (二)(105下、106下) 電腦圖學導論(105下、106上) 遊戲企劃與設計原理實習(106上) 電腦圖學導論實習(106上) 計算機圖學(106上) 遊戲程式設計(106下)
郭秉宸	7QUARK/製作人	電腦遊戲賞析(106上) 電腦圖學導論(106上) 3D電腦遊戲 (一)(106上) 3D電腦遊戲 (二)(106下) 遊戲程式設計(106下)
周軒廷	台灣盈米科技/專案助理	物件導向程式設計(106下、107下) 物件導向程式設計實習(106下、107下) 電腦圖學導論(107上) 電腦圖學導論實習(107上) 計算機圖學(107上) 遊戲程式設計(107下) 3D電腦遊戲 (一)(107下)

陳威光	ARvsVR工作坊/ 講師	3D電腦遊戲 (一)(104上、103上) 3D電腦遊戲 (二)(104下、103下)
羅應陞	英業達AI中心工 程師	3D電腦遊戲 (一)(108上) 電腦圖學導論(108上) 電腦圖學導論實習(108上)

B. 輔導學生之研究

● 碩士班歷屆指導學生

學生	論文題目	中/英	學年
鍾賢廣	基於霍爾效應的任意曲面上定位方法	中	107
林進仰	光學同調斷層牙周檢測系統	中	107
陳彥霖	利用擴散式曲線與四角網格重構法降低資料使用量	中	107
陳柏君	生物組織遮蔽下之牙結石偵測系統	中	107
羅應陞	利用自然光的即時室內照明控制	中	106
陳奕佑	基於骨架資訊並利用 N-旋轉對稱向量場之重新網格化	中	106
陳致成	互動式光學斷層牙齒掃描重建	中	106
李建緯	利用特徵網格分析之骨架感知重新網格化	中	105
郭鴻年	利用平滑線場網格化鑲嵌可縮放向量圖	中	105
翁世璋	利用最佳化各式參數實現漫畫框格切割	中	105
李政其	傻瓜運鏡：虛擬拍立得掌鏡系統	中	104
曾柏達	以圖像處理器平行加速大地電磁演算法	中	104

● 專題指導學生

學生	學年	專題題目
陳奕佑、蔡祖寧	104	程序化網點產生Procedure Screentone Generator
黃教荃	104	Brimo APP開發
羅應陞、陳彥霖	105	基於顯著性的圖形向量化與即時薄板曲線
林進仰、陳政煬、鍾賢廣	105	賽車遊戲系統與編輯工具
邱嘉興、黃弘文	106	VR釣魚
蔡中旗	106	虛擬智能操偶手套
范茗翔、鄭鈺哲	108	線上3D建模之貼圖工具開發
謝公曜	108	地景設計之2D曲線編輯工具開發
邱韋霖、謝宜杭	108	3D捕魚機場景設計工具開發

● 實習指導學生

學年	學生	公司
103學年暑假	陳奕佑、蔡祖寧、黃教荃	鈦象電子
104學年度	李政其、郭鴻年、葉致祥	美國加州杜比音效公司
104學年度	林進仰、陳政揚、鍾賢廣、陳彥霖、林德潔	
104學年暑假	羅應陞	鈦象電子
105學年暑假	朱駿采	鈦象電子
105學年	林進仰、邱嘉興、鍾賢廣、黃弘文、蔡中旗	鈦象電子
106學年	邱嘉興	日本東京大學
106學年暑假	羅應陞	英業達之AI中心
106學年暑假	范茗翔、謝公曜、邱韋霖	鈦象電子
107學年度	邱嘉興、郭俊廷	鈦象電子
107學年度	黃弘文	昇暘科技
107學年度	邱韋霖、謝宜杭	鈦象電子
108學年度	邱嘉興	日本東京大學

● 口試指導學生

學校	學生	學年
國立台灣大學	胡柯民	107
國立臺灣科技大學	張家瑋、郭皓程、林德潔、來國彥、鄭皓宇、呂仁傑	107
國立臺灣科技大學	Jonathan Hans Soseno、John Jethro Corpuz Virtusio	107
國立臺灣科技大學	呂柏儒、潘琮皓	107
國立臺灣科技大學	Ardiawan Bagus Harisa、劉宥銘	107
國立臺灣科技大學	李昆達	107
國立臺灣科技大學	詹竣傑、劉俊成、廖偉丞	106
國立臺灣科技大學	丁奕豪	106
國立臺灣科技大學	Hadziq Fabroyir	106
國立臺灣科技大學	陳建樺、陳柏安、顏天明、黃紹奕、張哲瑋	105
國立清華大學	林文勝、林佑恩、蕭任宸	105
國立臺灣科技大學	李亭緯、白勝宏	105
國立師範大學	黃浩庭、賴威豪、唐昌宇	105
國立臺灣大學	蔡佳昱、李維哲	105

國立臺灣大學	黃大源	105
國立臺灣科技大學	洪仕軒	104
國立臺灣科技大學	蘇柏巨	104
國立臺灣大學	官順輝	104

C. 教學績效

- ◇ 與姚智原、戴文凱和花凱龍教授合作，於108年度成立AI跨域應用產業碩士專班。
- ◇ 指導學生獲獎
 - 指導學生羅應陞與陳彥霖之專題「基於顯著性的圖形向量化與即時薄板曲線」獲得系上第一名，並參與2017年全國技專校院學生實務專題製作競賽入圍。
 - 指導學生蔡中旗與周紀愷之專題「虛擬智能操偶手套」獲得系上第一名，並參與2018年全國技專校院學生實務專題製作競賽暨成果展資工通訊群之第三名。

服務

A. 校內服務

● 國際訪問及招生

- ◇ 代表資訊創新中心訪問德國TU Dortmund、Hochschule Niederrhein Krefeld和RWTH Aachen University，2016/08
- ◇ 代表資訊創新中心訪問日本東京大學，2019/08

● 校內行政服務

- ◇ 院課程委員會委員(108)
- ◇ 系學生校外實習委員(108)
- ◇ 系課務暨招生委員會委員(107、108)
- ◇ 校科技權益委員會委員(108)
- ◇ 校產學合作委員(104)
- ◇ 校圖書館委員(107)
- ◇ 院空間規劃委員(104)
- ◇ 院務會議代表(105)
- ◇ 院務會議代表候補委員(106)
- ◇ 院發展委員會代表(105、107)
- ◇ 院四技實測委員(106)
- ◇ 系學術與系務委員會委員(106)
- ◇ 四年制導師(107、108)
- ◇ 電資不分系導師(104、105、106、107、108)
- ◇ AI跨域應用產業碩士專班出題(107)
- ◇ 博士班一般生(104、105、107)
- ◇ 博士、碩士甄試委員(104、105、106)
- ◇ 電資學院四年制甄試(104)
- ◇ 大轉學考稽核候補委員(106)
- ◇ 大學四技技優推甄委員(106)
- ◇ 大學四技推甄委員(104、107)
- ◇ 全校不分系四年制甄試(107)

● 其它服務

- ◇ 學校頂尖計畫及典範計畫之實際操作展演。
 - ◆ 104年度上學期典範期中展演。
 - ◆ 104年度下學期頂尖計畫期中展演。
 - ◆ 104年度教育部審查。
- ◇ 協助2015、2016、2017、2018和2019學校及電資學院外賓參訪參觀鈺象之互動娛樂研究中心。
- ◇ 參與協助2016、2017、2018和2019學校教育部深耕計畫之實際展示。
- ◇ 2015、2016、2017參與高中及高職招生宣導活動。
- ◇ 參與校內頂尖大學計畫
 - ◆ 104年度—創意母體中心之感知互動實驗室。
- ◇ 辦理104、105、106、107、108年度遊戲設計競賽—鈺象電子公司贊助100000元。
- ◇ 辦理107、108年度遊戲設計競賽—有夢最美贊助30000元。
- ◇ 2015、2017和2018與SONY簽定合作免費借用PS4開發機。
- ◇ 擔任台北商業大學創意設計競賽評審(105)

B. 校外服務

● 擔任國際會議之主辦

- ◇ 2016 Computer Graphics Workshop—Chair
- ◇ 2016 Pacific Visualization—Cochair

● 擔任國際會議之協辦

- ◇ 2014 Conference on Technologies and Applications of Artificial Intelligence—Demonstration Track Chair
- ◇ 2017 Pacific Graphics—Local coordinator Chair

● 重要支援國家產業審查活動

- ◇ 經濟部工業局小型企業創新研發計畫(SIIR)審查委員。
- ◇ 新北市文化局專案規畫審查委員。
- ◇ 新北市十三行博物館專案規畫審查委員及期中審查委員。
- ◇ 台北市職能發展學院107年虛擬及擴增實境產業應用座談暨委外承訓單位履約說明會委員。
- ◇ ITSA互動多媒體設計與整合應用組出題委員。

- **擔任重要國際期刊論文之審查委員**
 - ◇ IEEE Transactions on Circuits and Video Technology
 - ◇ IEEE Transactions on Vehicle Technology
 - ◇ IEEE Access
 - ◇ Computer Graphics Forum
 - ◇ Journal of Information Science and Engineering (JISE)
 - ◇ Computers & Graphics
 - ◇ Journal of Visual Communication and Image Representation

- **擔任重要會議論文之議程委員**
 - ◇ 2015、2016、2017和2108 Computer Graphics Workshop
 - ◇ 2017 Pacific Graphics Program Committee.
 - ◇ 2018和2019 ACM Symposium on Virtual Reality Software and Technology Program Committee.
 - ◇ 2018 Computer Vision, Graphics, and Image Processing (CVGIP).
 - ◇ 2016, 2017, 2018 ACM Symposium on Virtual Reality Software and Technology
 - ◇ 2018 NICOGRAPH International
 - ◇ 2018和2019 IEEE International Symposium on Multimedia
 - ◇ 2018 European Workshop on Visual Information Processing

- **國科會專題計畫審查**
 - ◇ 104和105學年度計畫審查：大專學生參與專題研究計畫-工程處專題研究計畫線上審查。
 - ◇ 104、105、106、107和108學年度計畫審查：工程處專題研究計畫線上審查。

- **近期受邀國內外學術研討演講和產業媒合**
 - ◇ 主辦單位：科技部
Date：2019/07/25
Title：可穿戴式行動輔具—手語翻譯手套與VR牙醫訓練
 - ◇ 主辦單位：東京大學資訊學系
Date：2018/02/28
Title：Screen-aware Manga Reader and Manga Vectorization
 - ◇ 主辦單位：Oregon State University
Date：2017/07/28
Title：Image Vectorization with TPS and SVG-embedded 3D Model Rendering.
 - ◇ 主辦單位：Portland State University
Date：2017/07/05
Title：NPR Chinese Painting Animation.

C. 服務績效

- ◇ 與姚智原和戴文凱教授合作，於107年8月續簽3年3000千萬，鈦象台科研發中心。
- ◇ 與姚智原、戴文凱和花凱龍教授合作，於107年度成立校級之資訊科技創新研究中心。
- ◇ 與姚智原老師及智財學院林瑞珠院長合作，參與台南歷史博物館AR虛擬人偶展覽計畫。
- ◇ 與姚智原老師及智財學院林瑞珠院長合作，參與台中國立美術館--20週年館慶活動，將藝術作品變成QR Code，有美麗的圖象和色彩，還能與畫作互動，並擴增實境，讓欣賞畫作變得更有趣。
- ◇ 參與十三行博物館之常設展VR互動展計畫。
- ◇ 參與文化部文資局—建構常設展「走入布袋戲」計畫。
- ◇ 參與新北市文化局—107年暑假之互動特展特展計畫。
- ◇ 參與新北市淡水古蹟博物館—滬尾礮臺互動特展計畫。
- ◇ 參與2017台北發明展。
- ◇ 參與2017烏鎮的互聯網大會。

教學效果—評量及學生作品





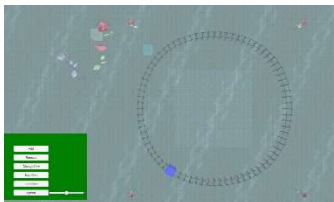


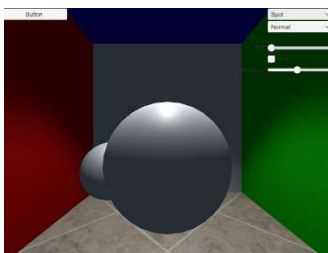
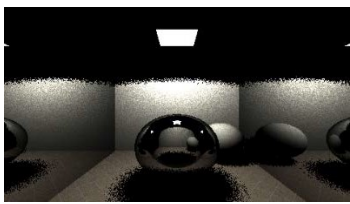
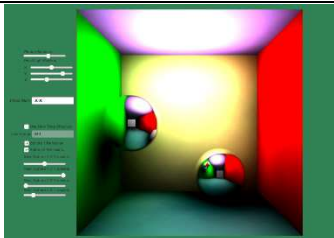

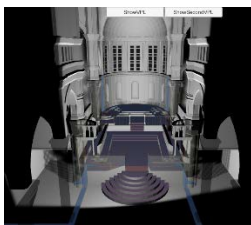
教材、教案、程式專案簡錄




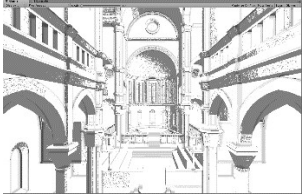

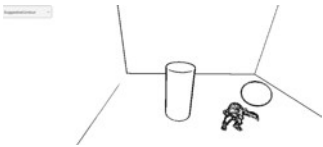
服務記錄

學生成果

教學及服務優良紀錄

2019 3D 遊戲設計(一)

作品名	作者	代表圖	作者	代表圖
Trains and Roller Coasters	林承達		郭俊廷	
Trains and Roller Coasters	陳弘展		黃欣云	
Trains and Roller Coasters	蔡奇霖			
Ray Tracer	林育生		林承達	
Ray Tracer	郭俊廷		黃欣云	
Ray Tracer	蔡奇霖			
Incremental Instant Radiosity Implementation	林承達		郭俊廷	

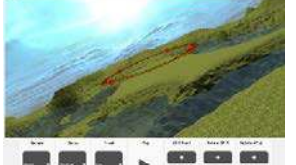
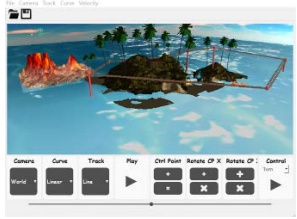

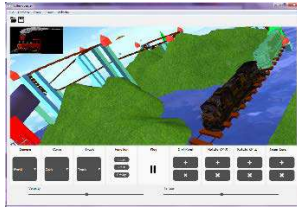
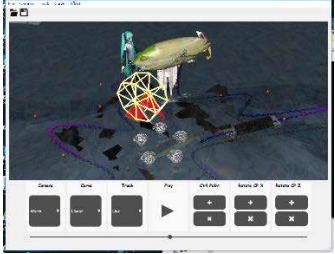
<p>Incremental Instant Radiosity Implementation</p>	<p>黃欣云</p>		<p>蔡奇霖</p>	
<p>Ambient Occlusion and Contour</p>	<p>林育生</p>		<p>林承達</p>	
<p>Ambient Occlusion and Contour</p>	<p>郭俊廷</p>		<p>陳弘展</p>	

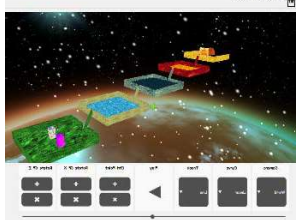

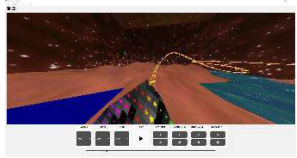
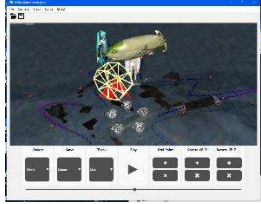
2019 Spring 遊戲程式設計節錄

作品名	作者	代表圖	作者	代表圖
迷宮冒險	柯名鴻、 蔡佳辰		吳育倫、 余柏葳	
迷宮冒險	周世賢、 張霆鋒		留希哲	
迷宮冒險	姚藝獎		孫上晏、 邱彥誠	
2D 遊戲	陳泳峰、 陳宥潤		陳賢張、 周錫緯	
2D 遊戲	林尚箴		林鼎傑、 溫勇威	
2D 遊戲	鄭永泰、 洪瑞陽		蔡宏駿、 袁瑋成	
Android Game	孫上晏、 邱彥誠		陳泳峰、 陳宥潤	

<p>Android Game</p>	<p>林奕君、 羅偉庭</p>		<p>姚藝獎</p>	
<p>Android Game</p>	<p>曾裕翔</p>		<p>陳賢張、 周錫緯</p>	
<p>Fps Game</p>	<p>陳昀諒、 紀凱議</p>		<p>蔡宏駿、 袁瑋成</p>	
<p>Fps Game</p>	<p>鄭永泰、 洪瑞陽</p>		<p>王詠生、 馬孝傑</p>	
<p>Fps Game</p>	<p>晉從歲、 留希哲</p>		<p>林尚箴</p>	


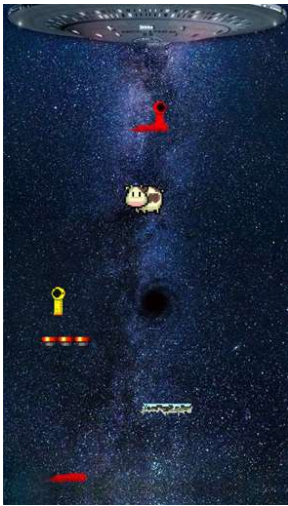

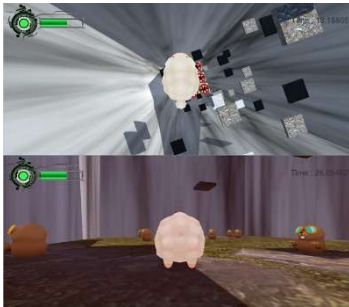





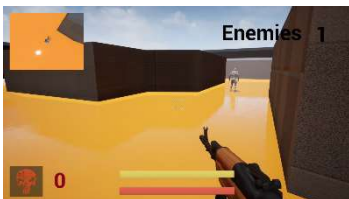
2018 Fall 圖學導論成果節錄

作品名	作者	代表圖	作者	代表圖
Amusement Park	晉從歲		姚藝獎	
Amusement Park	陳泳峰、 陳宥潤		林鼎傑	
Amusement Park	蔡宏駿、 袁瑋成		溫勇威	
Amusement Park	留希哲		陳洛翔、 韓悅華	
Amusement Park	陳靖升、 蔡富寬		鄭博安	
Amusement Park	林岳儒		林承達	

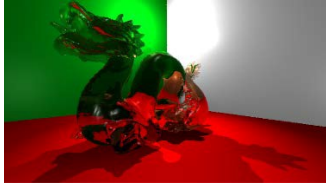

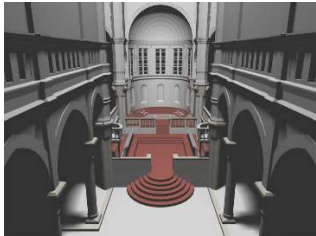
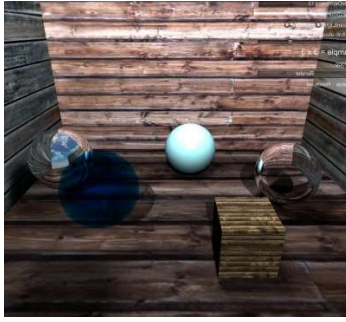

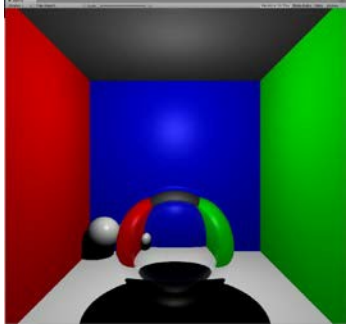


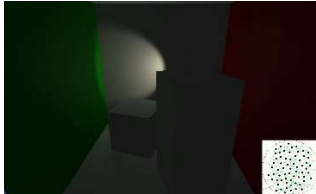

<p>Amusement Park</p>	<p>陳俊儒</p>		<p>林育生、 郭俊廷</p>	
<p>Amusement Park</p>	<p>廖瑄瑋</p>		<p>張子樂</p>	



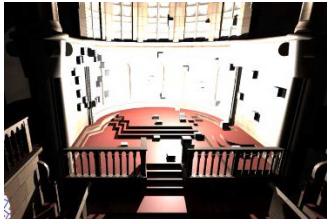
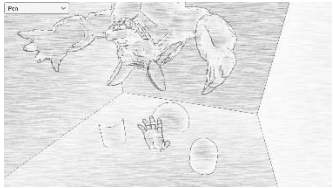
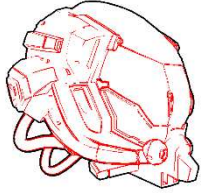

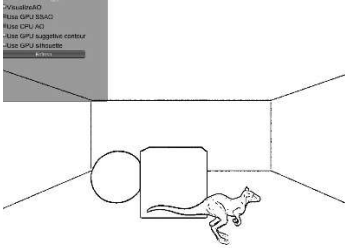
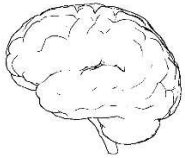
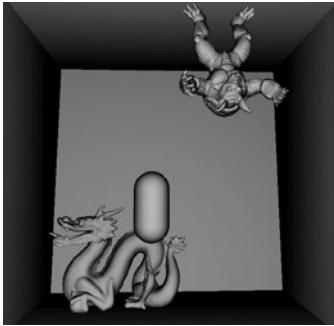
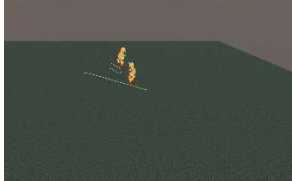
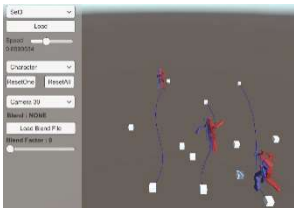
2018 Spring 遊戲程式設計節錄

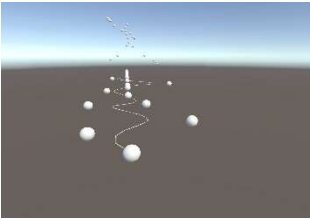
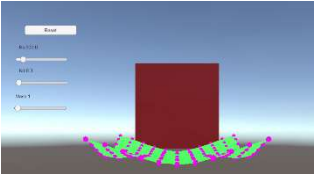
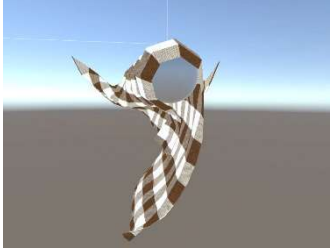
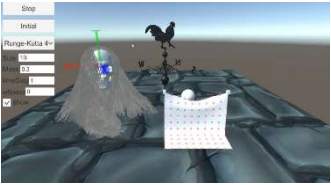
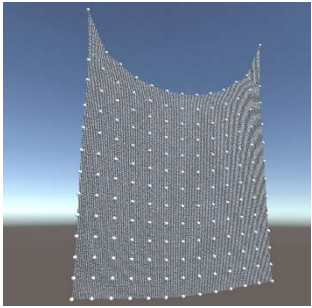
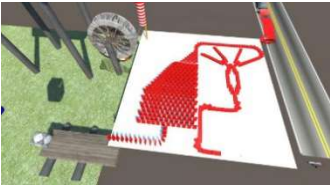
作品名	作者	代表圖	作者	代表圖
迷宮冒險	林科廷、 陳安泰		鄭鈺哲、 謝公耀	
迷宮冒險	陳羿凱、 謝宜杭		楊天慈、 謝宇庭	
迷宮冒險	林書宇、 楊駿焱		葉定豪、 石成峰	
戰車世界	黃小峰		張佑丞、 蘇莉恩	
戰車世界	李定鏞、 周政耀		屈家豪	
戰車世界	魏逢麟、 熊心平		林殷莊	
小朋友下樓梯	沈士豪、 許書豪		鍾洵丞、 吳毓書	

小朋友 下樓梯	吳冠穎		陳泳達、 陳牧凡	
小朋友 下樓梯	曾文謙、 邱韋霖		魏逢麟、 熊心平	
Fps Game	范茗翔、 王宥騰		顏佑庭、 周杰仕	
Fps Game	陳子揚、 王韋翔		葉定豪、 石成峰	
Fps Game	沈士豪、 許書豪		張博淞、 辜楷牧	

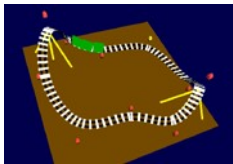
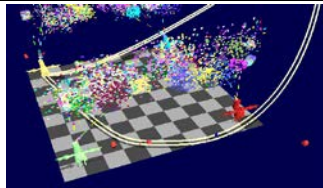
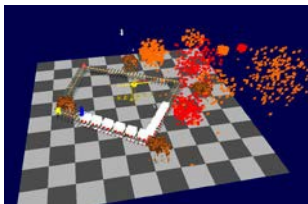
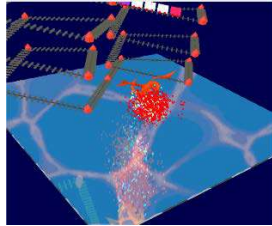
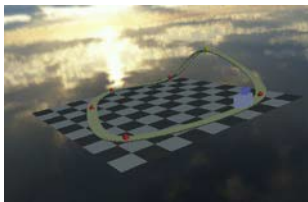
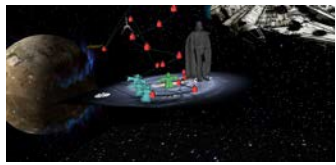
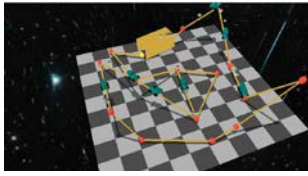
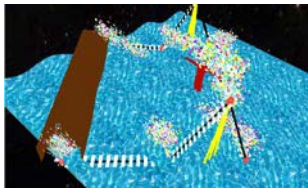


2017 3D Game

作品名	作者	代表圖	作者	代表圖
Ray Tracer	呂仁傑		林德潔	
Ray Tracer	邱嘉興		黃弘文	
Ray Tracer	劉益銓		蔡中旗	
Incremental Instant Radiosity Implementation	呂仁傑		林德潔	
Incremental Instant Radiosity Implementation	邱嘉興		黃弘文	




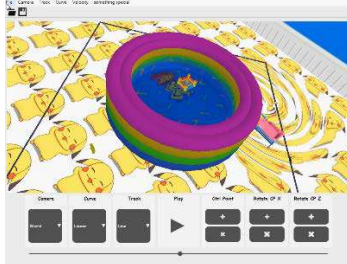




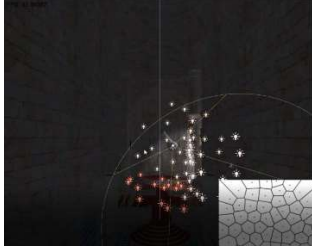
<p>Incremental Instant Radiosity Implementation</p>	<p>劉益銓</p>		<p>蔡中旗</p>	
<p>Incremental Instant Radiosity Implementation</p>	<p>鍾賢廣</p>			
<p>Ambient Occlusion and Contour</p>	<p>呂仁傑</p>		<p>林德潔 劉益銓</p>	
<p>Ambient Occlusion and Contour</p>	<p>邱嘉興</p>		<p>黃弘文</p>	
<p>Ambient Occlusion and Contour</p>	<p>蔡中旗</p>		<p>鍾賢廣</p>	
<p>Motion Path Editing</p>	<p>邱嘉興</p>		<p>黃弘文 蔡中旗</p>	

<p>Motion Path Editing</p>	<p>蔡宇彥</p>			
<p>Particle Simulation with Physics Engine</p>	<p>張哲寬</p>		<p>黃弘文 邱嘉興</p>	
<p>Particle Simulation with Physics Engine</p>	<p>蔡中旗</p>		<p>蔡宇彥</p>	
<p>Chain Reaction</p>	<p>邱嘉興 黃弘文 蔡中旗</p>			

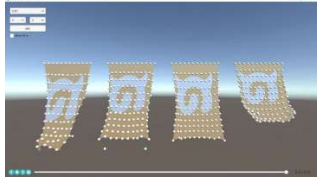

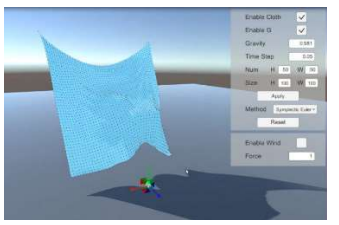

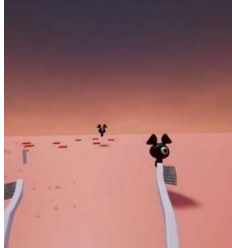
2017 Fall 圖學導論成果節錄

作品名	作者	代表圖	作者	代表圖
雲霄飛車	黃浩禎、 林書宇		邱韋霖	
雲霄飛車	王宥騰、 范茗翔		鄭鈺哲、 謝公耀	
雲霄飛車	謝宜杭		楊天慈、 謝宇庭	
雲霄飛車	葉定豪、 石成峰			
Amusement Park	黃浩禎、 林書宇		葉定豪、 石成峰	
Amusement Park	張哲寬			



2016 3D 遊戲設計

作品名	作者	代表圖	作者	代表圖
Trains and Roller Coasters on PS4	羅應陞			
Trains and Roller Coasters	郭昱祭		陳柏君	
Water Surface and Rendering	郭昱祭			
Chain Reaction	呂柏儒 陳柏君 郭昱祭			
Incremental Instant Radiosity Implementation	朱駿采 潘琮皓		呂仁傑	
Incremental Instant Radiosity Implementation	呂柏儒 陳柏君		林進仰 鄭皓宇	

Incremental Instant Radiosity Implementation	胡柯民		郭昱燊 林俞玘	
Incremental Instant Radiosity Implementation	陳奕佑		陳彥霖	
Incremental Instant Radiosity Implementation	鍾賢廣		羅應陞	
Motion Path Editing	朱駿采 潘琮皓		呂仁傑 鍾賢廣	
Motion Path Editing	呂柏儒 陳柏君		胡柯民	
Motion Path Editing	郭昱燊 林俞玘		陳奕佑	
Motion Path Editing	陳彥霖		羅應陞	
Particle Simulation with Physics Engine	朱駿采 潘琮皓		呂仁傑 鍾賢廣	

<p>Particle Simulation with Physics Engine</p>	<p>呂柏儒 陳柏君</p>		<p>胡柯民</p>	
<p>Particle Simulation with Physics Engine</p>	<p>郭昱燊 林俞玆</p>		<p>陳奕佑</p>	
<p>Particle Simulation with Physics Engine</p>	<p>羅應陞 陳彥霖</p>			
<p>3D Game</p>	<p>朱駿采 潘琮皓 林俞玆</p>		<p>呂柏儒 陳柏君 郭昱燊</p>	
<p>3D Game</p>	<p>羅應陞 胡柯民 鄭皓宇</p>			

2016 Spring 圖學導論成果節錄

<p>Amusement Park</p>	<p>蔡中旗</p>		<p>黃弘文</p>	
---------------------------	------------	---	------------	---

2015 3D 遊戲設計

作品名	作者	代表圖	作者	代表圖
PS4 Project	黃紹奕、 李建緯			