Graphics Programming in a Nutshell

AUTODESK MAYA

blender

AUTODESK 3DS MAX

Graphics Program

with shading
Graphics APIs

- **OpenGL** (Core, ES, Web)
  - Multi-platform graphics APIs

- **Direct3D** (DirectX9, DirectX11, DirectX12)
  - Only for Windows systems

- **Vulkan**
  - New generation of multi-platform graphics API

- **Metal**
  - Only for Apple systems
Graphics Programming

- **Low-level (OpenGL, DirectX, Vulkan, Metal)**
  - Direct call to API functions
  - Developed with C++, C#, Java, Python…

- **Mid-level**
  - Abstraction layer on top of graphics APIs
  - **Java3D, Qt3D, bgfx**

- **High-level**
  - Game engines
  - Visualization toolkits
GUI Programming

- Qt
- GTK+
- wxWidgets
- WPF (Windows only), ...
- Java GUI toolkits
- ...
- Low level system I/O utilities
  - SDL, FreeGLUT, GLFW
Visualization Toolkits

- ParaView
- VisIt
- Mayavi
- VolumeShop (developed @TU WIEN)
- MeVisLab
- + many more…

*Research groups often develop their own solution*
Game Engines

- Proprietary
  - Unity3D – Unity Technologies
  - Unreal Engine – Epic Games
  - CryEngine - CryTek
  - FrostBite – EA
  - Source 2 – Valve
  - + many more in-house engines

- Open source
  - Orgre, Torque3D, Panda3D, Irrlicht, MonoGame…
Why Unity3D?

- One of the most popular engine
- WYSIWYG editor
- Easy to use (C# scripting)
- Free
- Graphics API agnostic
- Cross-platform
- Extensive documentation
- Asset manager (textures, meshes, animations..)
- Built-in UI framework
- Multiplayer support
- …
Why Unity3D?

- Low-level graphics programming is too cumbersome
- Other game engines have steep learning curve
- Right balance between flexibility & ease-of-use
- Perfect for introduction to graphics programming

Caution!

Game engines are not the perfect all-in-one solution. For developing productivity softwares it might be better to simply start from scratch or using a visualization toolkit instead.
Unity3D Workflow

ASSET CREATION TOOLS

AUTODESK MAYA
blender
AUTODESK 3DS MAX

CODE EDITORS

Visual Studio
MonoDevelop

E. Gröller, J. Schmidt, M. Le Muzic
Unity3D Editor

**What You See Is What You Get** principle (WYSIWYG)

1 Scene View
2 Hierarchy
3 Inspector
4 Game View
5 Project
- Outline the content of the scene
- Hierarchical structure (parent, children)
- Easy manipulation of complex scenes
- Everything is a Game Object (lights, cameras, characters, objects, ...)
- Game Objects have components (mesh, audio, physics, script, etc.)
Game Objects

Game Objects may contain other Game Objects
Every Game Object has a transform
- Position, rotation, scale
- Only local properties are displayed
- Global properties are computed internally
Components are plugins that get attached to Game Objects.

- Components can be built-in (collider, mesh, renderer,...)
- Components can also represent a script...
Scripting

- Scripts are empty components (no built-in functionally)
- Must be attached to a game to run
- Can access transform and components properties
- Can also access other Game Objects in the scene
Useful stuffs

- `this.gameObject;` // The reference to the game object
- `this.transform;` // Position, rotation, scale of the game object
- `this.transform.parent;` // Transform of the parent object
- `this.GetComponent<Type>();` // Get component attached to game object
- `this.AddComponent<Type>();` // Attach component to game object
- `GameObject.Find(string name);` // Find game object in the scene

Useful callbacks

- `void Start () {}` // Called when the game starts to play
- `void Update() {}` // Called every time the view gets refreshed
- `void OnDestroy() {}` // Called when the game object is destroyed

... Many more useful stuffs, check the documentation online [http://docs.unity3d.com/ScriptReference](http://docs.unity3d.com/ScriptReference)
using UnityEngine;

// Inherits base class
public class CustomComponent : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }
}
using UnityEngine;

// Inherits base class
public class CustomComponent : MonoBehaviour
{
    public int myValue;

    // Use this for initialization
    void Start ()
    {
    }

    // Update is called once per frame
    void Update ()
    {
    }
}
Scripting
Game Objects and Components can also be created from a script

```csharp
using UnityEngine;

public class SpawnObject : MonoBehaviour
{
    // store gameObjects reference
    public GameObject parent;
    public GameObject child;

    void Start()
    {
        // spawn object
        child = new GameObject("Child Object");

        // Add Components (not compulsory)
        child.AddComponent<Rigidbody>();
        child.AddComponent<MeshFilter>();
        child.AddComponent<BoxCollider>();
        child.AddComponent<MeshRenderer>();

        child.transform.parent = parent.transform;
        Debug.Log(parent.transform.GetChild(0).name);
    }
}
```
In order to be rendered Game Objects need:

- Mesh Filter Component
- Mesh Renderer Component
Mesh Filter contains a reference to a mesh
Mesh Renderer contains a reference to a material
Meshes

- Mesh holds the internal data of a 3D model
- Indices, Vertices, Normals, UVs, Colors
- Meshes usually authored with Maya, Blender, etc., and then imported in Unity
- Support for all major file formats for meshes
  - .obj, .fbx, .stl, .ply, .x, .3ds, .dae, .ase, …
Meshes can also be created from script

```csharp
using UnityEngine;
public class MeshExample : MonoBehaviour
{
    public Vector3[] newVertices;
    public Vector2[] newUV;
    public int[] newTriangles;

    void Start()
    {
        Mesh mesh = new Mesh();
        GetComponent<MeshFilter>().mesh = mesh;
        mesh.vertices = newVertices;
        mesh.uv = newUV;
        mesh.triangles = newTriangles;
    }
}
```
Mesh Topology

- Vertices are 3D points in space
- Indices indicate how vertices are connected
- Mesh topology indicate the geometry type

```java
public void SetIndices(int[] indices, MeshTopology topology);
```
Materials

- Material define how meshes will be rendered
- Colors, textures, reflections, transparency, shadows,…
- Interface to the shader program
- Shading properties can be set manually via the editor
- Can also be set from script

```csharp
using UnityEngine;

public class ExampleClass : MonoBehaviour
{

    void Start()
    {
        Material material = GetComponent<MeshRenderer>().material;
        material.SetColor("_Color", Color.red);
    }
}
```
Shader Programming in a Nutshell

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Legacy Shaders

- Wide variety of legacy shaders
- Work out-of-the-box, no need to code them
Custom Shaders

- Possible to write custom shaders
- HLSL – cross compiled to GLSL internally for OpenGL platforms
- Modern shading functionalities on Desktop platforms
- More limited functionalities on other platforms (mobile, WebGL)
- Must start from scratch, e.g. perform lighting, texturing manually from code
Simple Color Shader

Shader "Custom/ColorShader"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            float4 vert(appdata_base v) : POSITION
            {
                return mul(UNITY_MATRIX_MVP, v.vertex);
            }

            float4 frag(float4 position:POSITION) : COLOR
            {
                return float4(1,0,0,1);
            }

            ENDCG
        }
    }
}

Pass 0
Shader "Custom/CustomShader"
{
    Properties
    {
        _MyColor("My Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            float4 _MyColor;

            float4 vert(appdata_base v) : POSITION
            {
                return mul(UNITY_MATRIX_MVP, v.vertex);
            }

            float4 frag(float4 position:POSITION) : COLOR
            {
                return _MyColor;
            }
            ENDCG
        }
    }
}
Different steps of the a complex shading process

http://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html
Custom Drawing Functions

- Mesh Renderer component ensures the drawing of the object internally
- Possible to issue the draw calls manually (useful for developing custom effects)
- Simply need a reference to the mesh and to the material
- Draw call issued after Unity has finished rendering the scene
- No out-of-the box shadows !!
Custom Drawing Function

- Game Object Callbacks:
  ```c
  void OnRenderObject() {}
  // Where custom draw calls must be issued
  ```

- Drawing Function
  ```c
  Graphics.DrawMeshNow(Mesh mesh, Vector3 position, Quaternion rotation);
  // Issue low-level draw call for a mesh
  ```

- Bind shader via `Material.SetPass(int pass)`
using UnityEngine;

// Attach this script to a Camera
public class ExampleClass : MonoBehaviour
{
    public Mesh mesh;
    public Material mat;

    public void OnRenderObject()
    {
        // set first shader pass of the material
        mat.SetPass(0);

        // draw mesh at the origin
        Graphics.DrawMeshNow(mesh, Vector3.zero, Quaternion.identity);

        // draw mesh at game object location
        Graphics.DrawMeshNow(mesh, transform.position, transform.rotation);
    }
}
Useful Links and References

- https://unity3d.com/learn/tutorials
- http://docs.unity3d.com/ScriptReference