

# Wrap Up

CSCI 4611: Programming Interactive Computer Graphics and Games

Evan Suma Rosenberg | CSCI 4611 | Fall 2022

This course content is offered under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license.

### **Reminder: Complete the SRT!**

Student Rating of Teaching - Fall 2022

CSCI 4611 001 Programming Graphics and Games; Instructional Staff: Bridger...

Live



### The SRT closes tomorrow.



### The optional bonus quiz is now available on Canvas.

### It is due on **Tuesday, December 20 at 6pm**.

Everyone has been issued a 0 on the bonus quiz by default. Canvas has been set to automatically drop your lowest quiz grade.

# **Recap: First Half of Semester**

- Structure of interactive graphics programs
- Points and vectors
- 2D/3D transformations
- Polygonal modeling and mesh data structures
- Texture mapping



# **Recap: Second Half of Semester**

- Hierarchical transformations and animated characters
- Lighting models
- Shader programs
- Projection and the virtual camera
- Ray casting for 3D user interfaces



### **Computer Graphics Toolkits**

- In this class, you gained experience using a web-based computer graphics toolkit that was created specifically for this course.
- You will find a very similar structure in almost all graphics libraries and game engines.
- If you know the core concepts (math library, meshes, shaders, textures) you can pick up almost any graphics toolkit and find the corresponding functions.
- Also, since the shaders use GLSL, you can write custom shaders for pretty much any toolkit that uses OpenGL.

# What else is out there? (Graphics APIs)

- G3D
- Irrlicht Engine
- OGRE
- Hoarde3D
- OpenSceneGraph (OSG)
- OpenSG Visualization Toolkit (VTK)
- Three.js

Babylon.js



https://casual-effects.com/g3d

# What else is out there? (IDE + Scripting)

This style of game engine is very popular. There are tons of tutorial videos and examples to help developers get started.

- Unity
- Unreal
- CryEngine

Godot





# Let's take a little tour of Unity.

FYI - you can download it free for personal use.

### 1. Find the 3D graphics primitives (cube, sphere, cylinder, etc.)



### 2. Find the Math library (Vector3, Matrix4, Transform, Ray, etc.)

		Manual	Scripting API	Search scripting	۹	unity.com →
Version: 2019.2-						C#
Scripting API	Vector3					
UnityEngine	struct in UnityEngine	/ Implemented in:Un	hityEngine.CoreModule			Leave feedback
<ul> <li>UnityEngine.Accessibility</li> </ul>						
<ul> <li>UnityEngine.Al</li> </ul>	Description					
+ UnityEngine.Analytics	Description					
<ul> <li>UnityEngine.Android</li> </ul>	Representation of 3D vectors and points.					
<ul> <li>UnityEngine.Animations</li> </ul>	This structure is used throughout Unity to pass 3D positions and directions around. It also contains functions for doing common vector operations.					
<ul> <li>UnityEngine.Apple</li> </ul>						
<ul> <li>UnityEngine.Assertions</li> </ul>	Besides the functions listed below, other classes can be used to manipulate vectors and points as well. For example the <u>Quaternion</u> and the <u>Matrix4x4</u> classes are useful for rotating or transforming vectors and points.					
🛨 UnityEngine.Audio						
UnityEngine.CrashReportHandler	Static Proper	ties				
<ul> <li>UnityEngine.Diagnostics</li> </ul>	otationropei					
<ul> <li>UnityEngine.Events</li> </ul>	back	Shorthand for	writing Vector3(0, 0, -1)			
🛨 UnityEngine.Experimental	down	down Shorthand for writing Vector3(0, -1, 0).				
+ UnityEngine.iOS	forward	forward Shorthand for writing Vector3(0,0,1)				
<ul> <li>UnityEngine.Jobs</li> </ul>						
🛨 UnityEngine.Lumin	ien	Shorthand for	writing vectors(-1, 0, 0)			
<ul> <li>UnityEngine.Networking</li> </ul>	negativeInfinity Shorthand for writing Vector3(float.NegativeInfinity, float.NegativeInfinity, float.NegativeInfinity).					
<ul> <li>UnityEngine.Playables</li> </ul>	one Shorthand for writing Vector3(1, 1, 1).					
UnityEngine.Profiling	positiveInfinity	Shorthand for	writing Vector3(float.Pc	sitiveInfinity, float.PositiveInfinity, flo	at.PositiveInfinity).	
<ul> <li>UnityEngine.Rendering</li> </ul>	right	Shorthand for	writing Vector3(1, 0, 0)			
<ul> <li>UnityEngine.SceneManagement</li> </ul>	1.917	Oharthand for				
UnityEngine.Scripting	<u>up</u>	Snorthand for	writing vector3(0, 1, 0).			
UnityEngine.Serialization	zero	Shorthand for	writing Vector3(0, 0, 0).			
UnityEngine.SocialPlatforms						
+ UnityEngine.Sprites	Properties					
UnityEngine.TestTools						

Returns the length of this vector (Read Only)

UnitvEngine.TextCore

Inspector							
👕 🔽 Cube			🥅 Static 🔻				
Tag Untage	jed 🛛	🗘 Layer 🛛 De	fault ‡				
🔻 🙏 🛛 Transfo	rm		🔯 🗟 🔅				
Position	× 0	Υ 0	Z 0				
Rotation	× 0	Υ 0	Z 0				
Scale	X 1	Y 1	Z 1				

### 3. Find the material properties (colors, images, textures, etc.).





### 4. Find the shaders.

🚭 unity   documentation	Manual Scripting API Search manual Q unity3d.com →					
′ersion: 2019.2 -	Language : English 🗸					
Unity Manual	Unity Liear Manual (2010.2) / Crambins / Crambins Quantiew / Materiale Shadare & Tayturae / Writing Shadare					
<ul> <li>Unity User Manual (2019.2)</li> </ul>	Unity User Manual (2013.2) / Utaphiles / Utaphiles Over them / Imaterials, Sinders & rectores / Whiting Unders					
🛨 Packages	÷ >					
Working in Unity						
+ Importing	Writing Chadara					
+ 2D	writing Shaders					
Graphics	Shaders in Unity can be written in one of three different ways:					
<ul> <li>Graphics Overview</li> </ul>	Surface Shaders					
+ Lighting	Surface Shaders are your hest option if your Shader needs to be affected by lights and shadows. Surface Shaders make it easy to write compley					
+ Cameras	Shaders in a compact way - it's a higher level of abstraction for interaction with Unity's lighting pipeline. Most Surface Shaders automatically support					
Materials, Shaders & Textures	both forward and deferred lighting. You write Surface Shaders in a couple of lines of Cg/HLSL, and a lot more code gets auto-generated from that.					
Textures	Do not use Surface Shaders if your Shader is not doing anything with lights. For post-processed effects or many special-effect Shaders, Surface					
<ul> <li>Creating and Using Materials</li> </ul>	Shaders are a suboptimal option, since they do a bunch of lighting calculations for no good reason.					
<ul> <li>Standard Shader</li> </ul>	Vertex and Fragment Shaders					
<ul> <li>Standard Particle Shaders</li> </ul>						
<ul> <li>Physically Based Rendering Material Validator</li> </ul>	Vertex and Fragment Shaders are required if your Shader doesn't need to interact with lighting, or if you need some very exotic effects that the Surface Shaders can't handle. Shader programs written this way are the most flexible way to create the effect you need (even Surface Shaders are automatically					
<ul> <li>Accessing and Modifying Material parameters via script</li> </ul>	converted to a bunch of Vertex and Fragment Shaders), but that comes at a price: you have to write more code and it's harder to make it interact with lighting. These Shaders are written in Cg/HLSL as well.					
Writing Shaders	Fixed Function Shaders					
<ul> <li>Legacy Shaders</li> </ul>	Fixed Function Shaders are legacy Shader syntax for very simple effects. It is advisable to write programmable Shaders, since that allows much more					
<ul> <li>Video overview</li> </ul>	flexibility. Fixed function shaders are entirely written in a language called ShaderLab, which is similar to Microsoft's .FX files or NVIDIA's CgFX.					
<ul> <li>Terrain Engine</li> </ul>	Internally, all Fixed Function Shaders are converted into Vertex and Fragment Shaders at shader import time.					
<ul> <li>Tree Editor</li> </ul>						
<ul> <li>Particle Systems</li> </ul>	ShaderLab					
<ul> <li>Post-processing overview</li> </ul>	Regardless of which type you choose, the actual Shader code is always wrapped in ShaderLab, which is used to organize the Shader structure. It looks					
<ul> <li>Advanced Rendering Features</li> </ul>	like this:					
Procedural Mesh Geometry						

	Folder		Create	>	▼ Layout ▼	
Hierarchy	C# Script		Show in Explorer		à •=	
G SampleScen ♥ Main Camer ♥ Directional I ♥ Cube	Shader	>	Standard Surface Shader		¢,	
	Testing	>	Unlit Shader		Open	
	Playables >		Image Effect Shader			
	Assembly Definition		Compute Shader	lt+Ctrl+C		
	Assembly Definition Reference		Shader Variant Collection			
	TextMeshPro	>	Import New Asset			
	Scene		Import Package	>		
	Prefab Variant		Export Package			
	Audio Mixer		Find References In Scene			
	Material		Select Dependencies			
	Kateria Lens Flare Render Texture Lightmap Parameters		Refresh	Ctrl+R		
			Reimport			
			Reimport All			
	Custom Render Texture		Extract From Prefab			
	Sprite Atlas Sprites >		Run API Updater			
			Undate IIIElements Schema			
	Animator Controller Animation Animator Override Controller					
			Open C# Project			
	Avatar Mask					
	Timeline					
	Signal					
	Physic Material					
	Physics Material 2D					
	GUI Skin					
	Custom Font					
	Legacy	>				
	UIElements Editor Window					
	Brush					
	Terrain Layer					

### Notes about writing custom shaders

- Game engines will come with a variety built-in shaders, and you could potentially write an entire game without writing a line of shader code.
- However, many games will implement custom shaders to implement visual effects or achieve a specific aesthetic.
- Writing your shaders from scratch is often easier because modifying the built-in shaders, because the professional toolkits will often have very complex shader code that include normal mapping, reflections, transparency, and other effects integrated into one unified shader.

Shaders will generally be written in GLSL (OpenGL), HLSL (DirectX), or sometimes a custom format that is a high-level wrapper around one of the two.

### **GLSL vs. HLSL**

#### HLSL (Unity, Unreal)

```
//Vertex Shader
void MainVertexShader(
   float4 InPosition : ATTRIBUTE0,
   float2 InUV : ATTRIBUTE1,
    out float2 OutUV : TEXCOORDO,
   out float4 OutPosition : SV POSITION
   OutPosition = InPosition;
   OutUV = InUV;
}
//Fragment Shader
void MainPixelShader(
   in float2 UV : TEXCOORDO,
   out float4 OutColor : SV_Target0
   OutColor = float4(UV, 1.0, 1.0);
```

#### GLSL (Three.js, Godot, G3D, countless others)

```
//Vertex Shader
in vec4 InPosition;
in vec2 InUV;
out vec2 UV;
void main()
{
  gl_Position = InPosition;
  UV = InUV;
}
```

//Fragment Shader
in vec2 UV;

```
void main()
```

```
gl_FragColor = vec4(UV, 1.0, 1.0);
```

### 5. Find the higher-level things we have learned in the assignments:

- Initialization, update, and draw methods: You will always find methods where you can initialize the graphics, create objects in the scene, and implement code that needs to execute once per frame as needed for animation, etc.
- **Physics simulation:** You can always do it by hand using points, vectors, etc. Some game engines will also have a built-in physics engine to handle collision detection, gravity, bouncing, etc.
- **Hierarchical transformations:** Every toolkit will have some ability to organize models in the scene in a hierarchy. Unity has a graph of GameObjects, where each has a parent and a list of children.
- **Character animation:** Many toolkits will support some form of character animation using hierarchical transformations, like we've done. You will need to consider the same concepts we have in class (blending between mocap motions, etc.).
- **Camera and user interface:** There is often a mode for drawing 2D text and lines on top of the screen, like the 2D strokes in the final assignment. Sometimes this includes pre-defined routines for drawing buttons, sliders, etc. You will always find a Camera class or something similar that allows you to create 3D pick rays from the mouse position to create 3D user interfaces.

## Some additional game engine features

**Fancy real-time rendering effects:** Most are just extensions of what you have learned about using textures inside shaders and adjusting vertex positions and lighting calculations. Examples include normal mapping, bump mapping, reflections, "moving textures" like water, transparency effects, non-photorealistic effects, and more.

Al: Crowd simulation, flocking behaviors, natural language processing, terrain generation, etc.

**VR support:** The standards for VR/AR/XR are still actively being developed in many engines. Honestly Unity's API for this is still not that great and often changes between major versions, but it is really cool that if you are not doing anything too fancy in terms of the UI, it is actually quite easy to deploy to a PC-based VR system or mobile VR headset.

Let's discuss two final graphics math concepts that I have seen confuse students when working with game engines such as Unity.

### Watch out for Left-handed vs. Right-handed



- If I had to guess, I would estimate that 90% of graphics toolkits use a right-handed coordinate system.
- Unity happens to be in the 10% that picked left handed. *Ugh.*
- Thankfully, all the math really works out the same either way. Just switch to using the "left hand rule" rather than the "right hand rule" for cross products.
- Also be careful when importing models from 3D modeling software or connecting to other systems outside of Unity, like VR tracking systems or your phone's inertial tracker. Almost everybody else uses right handed coordinates.
- You can convert points and vectors from right-handed to left-handed by negating the one of the coordinates. (When Unity loads 3D models stored in right-handed format it usually automatically negates the X coordinate.)

Converting transformation matrices that include rotations is a bit trickier but can also be done.

### Watch out for Rotations

Different game engines will also adopt different conventions for how to represent rotations.

3D Rotations can be represented in several different ways:

- Euler angles (easiest to talk about, but several problems)
- Rotation matrices
- Quaternions (best for interpolation)

### Watch out for Rotations

Unity uses all three representations for rotations:

- **Euler angles** are first shown in the editor.
- Each frame, they are converted to **quaternions** to store in a GameObject.
- Programmers can also convert them to **rotation matrices**, apply a series of transformations in a script, and then convert it back to a quaternion.

# Some additional opportunities for those interested in computer graphics and games.

(unless you are imminently about to graduate, in which case congratulations!)

### Courses that can build on what you have already learned

### • CSCI 5607: Computer Graphics 1

Typically offered once per year in the Spring (taught by Victoria Interrante). This course is more theoretical/mathematical than this class.

### • CSCI 5611: Animation and Planning in Games

Typically offered once per year in the Fall (taught by Stephen Guy).

### • CSCI 5609: Visualization

Typically offered once per year in the Spring (taught by Dan Keefe).

### • CSCI 5619: Virtual Reality and 3D Interaction

Typically offered once per year in the Fall (usually taught by me).

### • CSCI 8980: Special Topics Courses

These are special one-time courses. Advanced undergraduates can take these classes with instructor permission!

## Getting involved in a research lab

• Research is an opportunity to more deeply explore topic areas that interest you.

You don't need to be a graduate student to take advantage of this!

- Building a mentorship connection with a professor. It's also a fantastic way to get very strong recommendation letters!
- Explore whether graduate school is right for you.
- You never know what you might discover!

This was my origin story... being a computer science professor was not on my radar until I had a summer research opportunity.

### How do I get started?

# • It's never to early to talk to a professor in a research area that interests you!

Undergraduates are often active participants in my lab, and I am currently advising several undergraduate research projects.

• This can often lead to funding from the Undergraduate Research Opportunities Program (UROP).

You can get a scholarship stipend to support a research project.

https://ugresearch.umn.edu/opportunities/urop

## **Final Thoughts**

I had a lot of fun teaching this course, and I hope you had fun taking it too!

If you are interested in chatting about opportunities to get more experience with computer graphics, virtual reality, and/or research, please feel free to reach out!

# **Final Thoughts**

I had a lot of fun teaching this course, and I hope you had fun taking it too!

If you are interested in chatting about opportunities to get more experience with computer graphics, virtual reality, and/or research, please feel free to reach out!



For those who are wondering about my video game plans over winter break!

