Game Engines
Creating Immersive Worlds, Fall 2018

What is a Game Engine?

So we can just head to Wikipedia for a definition, right:

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices, and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games[1] or to make it easier to port games to multiple platforms.
Game engines essentially provide a framework for you to rapidly build upon. Computers are incredibly stupid, so they need to be told exactly what to do, which means someone needs to write a very specific set of functions and methods describing how to draw a point on the screen, or how to play a sound, or what to do when you scroll your mousewheel down.

For example, this APPLE Basic code will draw a Fibonacci spiral (do not read it, just appreciate the length). This is just the code, which has to get read by another program (a compiler) before it can do anything useful at all.

```
10 HGR2 : CLEAR : HCOLOR=1
20 REM HPL0T 0,96 TO 279,96 : REM uncomment to draw axis OX
30 REM HPL0T 140,0 TO 140,191 : REM uncomment to draw axis OY
40 HCOLOR=2 : REM purple
50 X = 0 : Y = -1 : REM initial coordinates
60 D = 0 : REM direction
70 M = 3 : REM scale
80 CX = INT(288/2)-0.5 : CY = INT(192/2)-0.5
```
100 A = 0 : B = 1 : REM initial Fibonacci numbers
110 C = B : GOSUB 300
120 A = A + B
130 GET W : IF E > 0 GOTO 10
140 C = A : GOSUB 300
150 B = A + B
160 GET W : IF E > 0 GOTO 10
170 GOTO 110
180 END

300 REM draw square size C by C with starting corner at (X,Y)
310 IF D = 0 THEN DX = 1  : DY = 1
320 IF D = 1 THEN DX = -1 : DY = 1
330 IF D = 2 THEN DX = -1 : DY = -1
340 IF D = 3 THEN DX = 1  : DY = -1
350 DX = DX * C : DY = DY * C
360 GOSUB 400 : REM comment this line to just draw the spiral
370 R = C
371 IF D=0 THEN X1 = X      : Y1 = Y + DY
372 IF D=1 THEN X1 = X + DX : Y1 = Y
373 IF D=2 THEN X1 = X      : Y1 = Y + DY
374 IF D=3 THEN X1 = X + DX : Y1 = Y
375 GOSUB 600
380 X = X + DX : Y = Y + DY
385 D = D + 1
390 RETURN

400 ONERR GOTO 470 : REM drawing next square
410 HPLLOT X * M + CX, Y * M + CY
420 HPLLOT TO (X + DX) * M + CX, Y * M+ CY
430 HPLLOT TO (X + DX) * M + CX, (Y + DY) * M + CY
440 HPLLOT TO X * M + CX,  (Y + DY) * M + CY
450 HPLLOT TO X * M + CX,  Y * M + CY
460 GOTO 500
470 E = 1 : REM error (outside of screen)
500 RETURN

600 REM draw arc, centered at X1,Y1, radius R, quarter of circle.
610 ONERR GOTO 710
620 PI = 3.14159
630 RAD = PI * (D - 1)/2
640 HCOLOR=3 : REM white
650 HPLLOT CX + M * (X1 + COS(RAD) * R), CY + M * (Y1 + SIN(RAD) * R)
660 FOR G = RAD to RAD + PI/2 STEP 0.05
670 HPLLOT TO CX + M * (X1 + COS(G) * R), CY + M * (Y1 + SIN(G) * R)
680 NEXT
690 HCOLOR=2 : REM return to purple
700 GOTO 720
710 E = 1 : REM error (outside of screen)
720 RETURN

So you write all of this in 1990, and you get:
Back in the 1980s there were a number of construction sets that took a certain framework, or game play type, and made it easier for people to mashup and re-create, or 'mod' the game.

Fast forward to 1993, where we saw for the granddaddy of the FPS (first-person shooters)- *Doom*. The series focuses on the exploits of *Doomguy* an unnamed space marine.
It had multiplayer support, tons of gore, and best of all...it allowed player-created modifications and maps.

Doom came with a map editor that allowed you to create your own levels.

This allowed a whole generation of emerging gamers to also create their own worlds using the game engine's technology, so they didn't need to manually program various subsystems. All they simply had to do was create content and code, and the engine would take care of the lower-level tasks for them.
The id Tech 1 Engine, the engine behind Doom went open-source in 1993. This further allowed people to experiment (now you have access to all of the underlying code, not just what the game engine exposed)

Back to Game Engines

At the core, a game engine includes a subsystem that handles things like ‘rendering’—e.g., the graphics on the screen. It also includes a subsystem to handle complex calculations, such as physics and collision.

Speaking of physics engines, here is one of my favorite physics systems, from Grand Theft Auto 4: The Euphoria Game Engine:

Game engines allow us to re-use many common media and interaction paradigms and mediums and save us the trouble of implementing our own specific subsystems (rendering, audio, physics, AI, etc…). Modern game engines are basically a mashup of a number of separate engines, conveniently designed to work well together:
• **Rendering Engine**: Makes the graphics

• **Physics and Collision Engine**: Handles physics calculations and collisions

• **Animation Engine**: Character animations and scrip-table behaviors

• **Audio Engine**: Dynamic playback (including simulation and effects)

• **Scripting Engine**: Allows programmable interactions and gameplay

• **AI Engine**: Artificial agent intelligence and behaviors

• **Networking Engine**: Built-In support for networking multiple game clients

• **Build Engine**: Compiles one version of a game so that it can run on multiple platforms (Unity can publish to 25 platforms!)

---

**What is UE4**

We are going to crash right into one of the most advanced game engines humans have ever created and talk about what makes it a good tool for game development. This is a huge topic, so to begin I just want to highlight a few of its key differentiating features.

There are a number of reasons I’ve chosen for us to use Unreal Engine. Without sounding like an advertisement, it’s one of the most state-of-the-art game engines available to the public today, it’s open source, you can download the code on Github, it supports VR out of the box, has great rendering and graphical capabilities, it allows you to create large, networked open worlds (it’s the engine that powers Fortnight and PUBG), you can add attenuated spatially accurate sound. You can even use it to create cinematic in-game videos.

Unreal Engine was created by [Tim Sweeney](https://www.unrealengine.com), who, by the way, also created the video game ZZT, which was one of the first games (and game engines) I ever played:
Of course today, Tim’s game engines are a bit more advanced…

We’ll use all of these things throughout the course, and honestly, these are all features that we should come to expect in a modern mass-market game engine. Unity, a competing game engine that’s used in billions of games can do all of these things, but it requires a bit more tweaking out of the box to get the same results that UE4 provides immediately. Unity is particularly great at helping you develop mobile games, but has also been used extensively to create some incredible VR experiences. Many of which we will be playing over the course of the course. I personally use both engines, and if time permitted, I’d be encouraged to cover both engines. They are both great tools to help you realize your creative vision.

**Why UE4?**

Well, the feature that really sets UE4 apart, in my opinion, is it’s blueprints system. This system allows pros and beginners alike to rapidly prototype and even fully develop some very incredible interactions and mechanics, all without coding a single line of c++.

**Blueprints**
Blueprints are a visual scripting system that allows you to visually see the structure of your code and logic and allow you to reuse objects throughout our experiences.

They are one of two fundamental ways we can add interaction and life to our worlds in UE4 and both will generally get us the same results.

First, we could code, using a programming language called c++ and an integrated development environment like Visual Studio on the PC, or xcode on the Mac. Alternatively, Blueprints allow us to visually program our logic and data structures by connecting nodes together. Blueprints are pretty amazing, especially when combined with an understanding of object oriented programming and the concepts that underpin the c++ language, which we will a lot as we use the BP system.

So let's say we want to create a game where the player is in charge of a zombie retirement home and has to take care of a bunch of zombies in their old age. Your task for today is to build the scoring system. We have our zombie character done, but we want to have them call out when they are hungry for brains.

With an object-oriented structure, you can create one parent zombie class blueprint that contains all of the variables and functionality that are shared by all of your ‘zombies’, such as hunger, speed, etc…and then create extensions of that parent blueprint class for different types of zombies that only have the functions and variables available to that zombie, such as the ability to sprint or a function called climb on the ceiling for an especially, nope, not cool.

These are core properties of inheritance and re-usability that make object-oriented programming so powerful, and Unreal Engine's blueprint system gives non-programmers this same ability. So by using blueprints, we're going to be able to prototype rapidly, but also learn core concepts and frameworks that can apply to a variety of programming languages should you chose to take those up in the future.