CS354P

DR SARAH ABRAHAM

GAME ENGINE ARCHITECTURE

Assembly-CSharp - Assets\ScriptGame\scriptPlayer.cs -

File Edit View Search Project Build Run Versi

Globals.cs × GameControl.cs × Gamepl

scriptPlayer ▸ downPress ()

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class scriptPlayer : M
5      bool going, spotReached,
6      public short direction, s
7      movePlayer moveP; GameObj
8      Color colorStart; Color c
9
10     void Start() {// Use this
11       going = false; north =
12       speed = Globals.pSpeed;
13       moveP = GetComponent<mo
14       gameSFX = GameObject.Fi
15       gameBGM = GameObject.Fi
16       colorStart = renderer.m
17     }
18
19     void Update() { if(!Globa
20       //check if still moving
21       if(going){ //moving to
22         if(spotReached){
23          if(north){ direction = 1; transform.eulerAngles = new Vector3(0, 0, 0); }
24          if(east){ direction = 2; transform.eulerAngles = new Vector3(0, 0, 270); }
25          if(south){ direction = 3; transform.eulerAngles = new Vector3(0, 0, 180); }
26          if(west){ direction = 4; transform.eulerAngles = new Vector3(0, 0, 90); }
27         }
28         spotReached = false; if(steps == 3) sfxScript.sndMove();
29         //move in direction
30             //count steps until in next space
31         if(Globals.readyP) moveP.Move(direction); steps++; if(steps >= (20/speed)){ steps = 0; spotReached = true; }
32       }
33       else{ //let go of button
34         if(!spotReached){ //let go of button, but still moving
35           if(Globals.readyP) moveP.Move(direction); steps++; if(steps >= (20/speed)){ steps = 0; spotReached = true; }
36         }
```

Ready 57 : 59 INS Feedback Task List

BLUEPRINT

WHAT IS A GAME ENGINE?

- Low-level architecture
  - 2D/3D graphics system
  - Physics system
  - GUI system
  - Sound system
  - Networking system
- High-level architecture
  - Game objects
  - Game mechanics
- Toolsets
  - Level editor
  - Character and animation editor
  - Material creator
- Subsystems
  - Run-time object model
  - Real-time object model updating
  - Messaging and event handling
  - Scripting
  - Level management and streaming

RUN-TIME SYSTEM

- Low-level architecture
  - 2D/3D graphics system
  - Physics system
  - GUI system
  - Sound system
  - Networking system

SYSTEM MODULARITY FOR PLAY

- Keep systems as independent as possible during run-time
  - What does this mean and how do we do this?
- Examples of keeping systems independent:
  - The scene still renders even if the physics engine fails
  - The world state is consistent between client and server even if sounds or animations are lost
  - The game loop does not wait for AI to make a decision

SYSTEM MODULARITY FOR DEVELOPMENT

- Keep systems as independent as possible during development
  - What does this mean and how do we do this?
- Examples of keeping systems independent:
  - The game is playable before the GUI is built
  - Changes a programmer makes do not clobber the artist or designer pipelines
  - The binary for a game that doesn't use physics does not require the physics libraries

HIGH-LEVEL ARCHITECTURE

- Game objects
- Game mechanics

MODELING DATA

- What sort of data is in a game and what systems need to use this data?
- Data must be passed between various run-time systems in an efficient manner!
- Two broad approaches
  - Object-centric
  - Property-centric
- The choices made here will have ramifications for every single subsystem and any communication between subsystems!

WORKING WITH OBJECTS

- Use of classes (attributes and behaviors) to create and update data
- Engine defines run-time systems and supporting systems within its own frameworks of classes
- Game developer extends these classes through inheritance to match specific behavior required

WORKING WITH PROPERTIES

- Use of tables of properties and object ids to define and update data
- Engine defines run-time systems and supporting systems within its own frameworks of API calls
- Game developer passes “object” information required by systems to exhibit correct behavior

WHAT DOES THIS MEAN FOR DEVELOPMENT?

- Object-centric approaches have a more rigid structure
  - Much upfront mastery required
  - Better debugging tools longer term
- Property-centric approaches have a more fluid structure
  - Easier early prototyping
  - Potentially confusing structures in large-scale projects

UNREAL ENGINE

- UE5 is object-oriented and uses **components** and **interfaces** extensively
  - Large codebase with *many* specific functionalities
- Must understand the underlying architecture to work effectively in it!

TOOLSETS

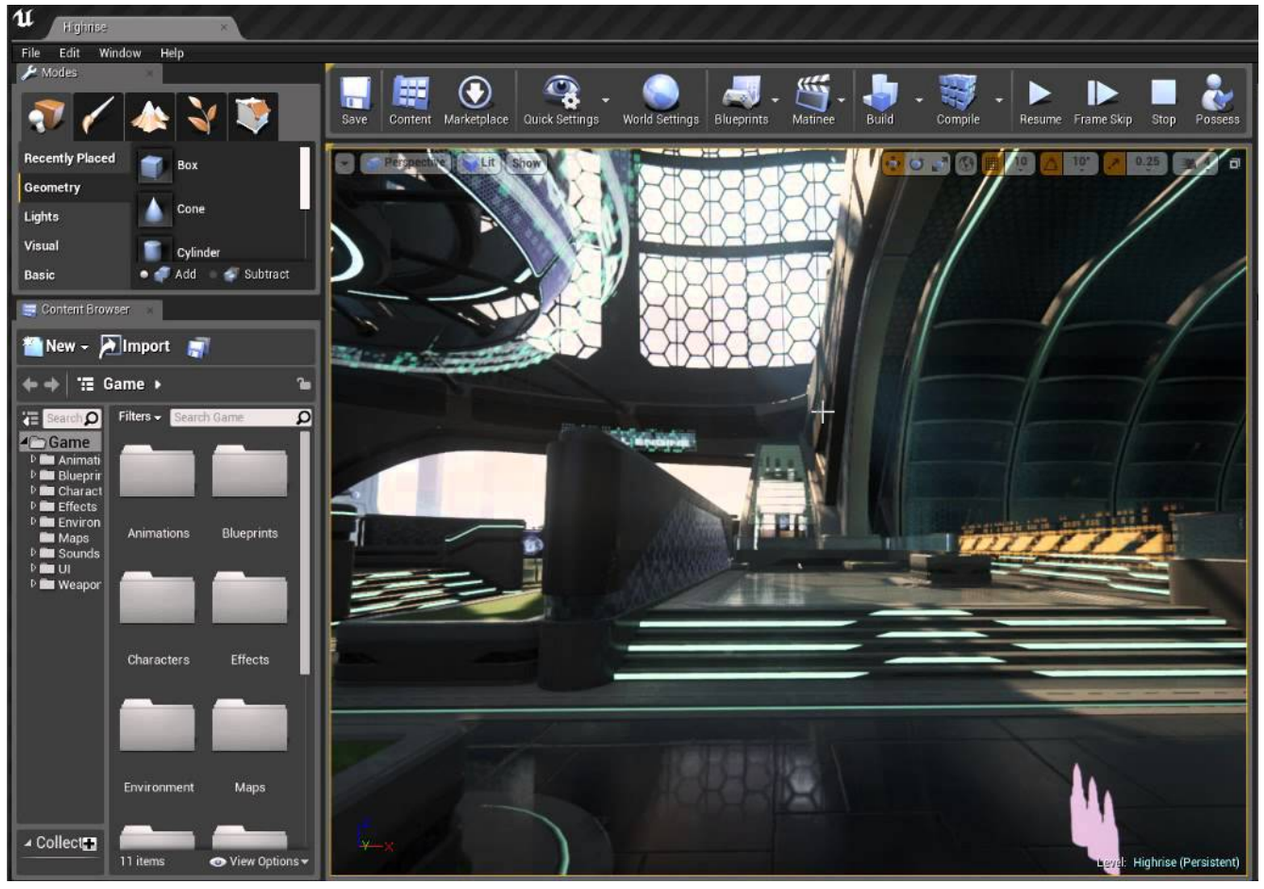
- Level editor
- Character and animation editor
- Material creator

DESIGNER TOOLS

- Tools related to game design depend heavily on the game
  - Crafting/leveling systems may primarily be done in CSVs
  - Combat/movement systems closely tied to in-game animations and physics systems
  - Dialogue usually written externally then imported
- Game engines may or may not support any of these natively

LEVEL EDITORS

- Provided by most engines
- May or may not generate level content programmatically/ procedurally
- Editor considerations also include loading/streaming/ level of detail

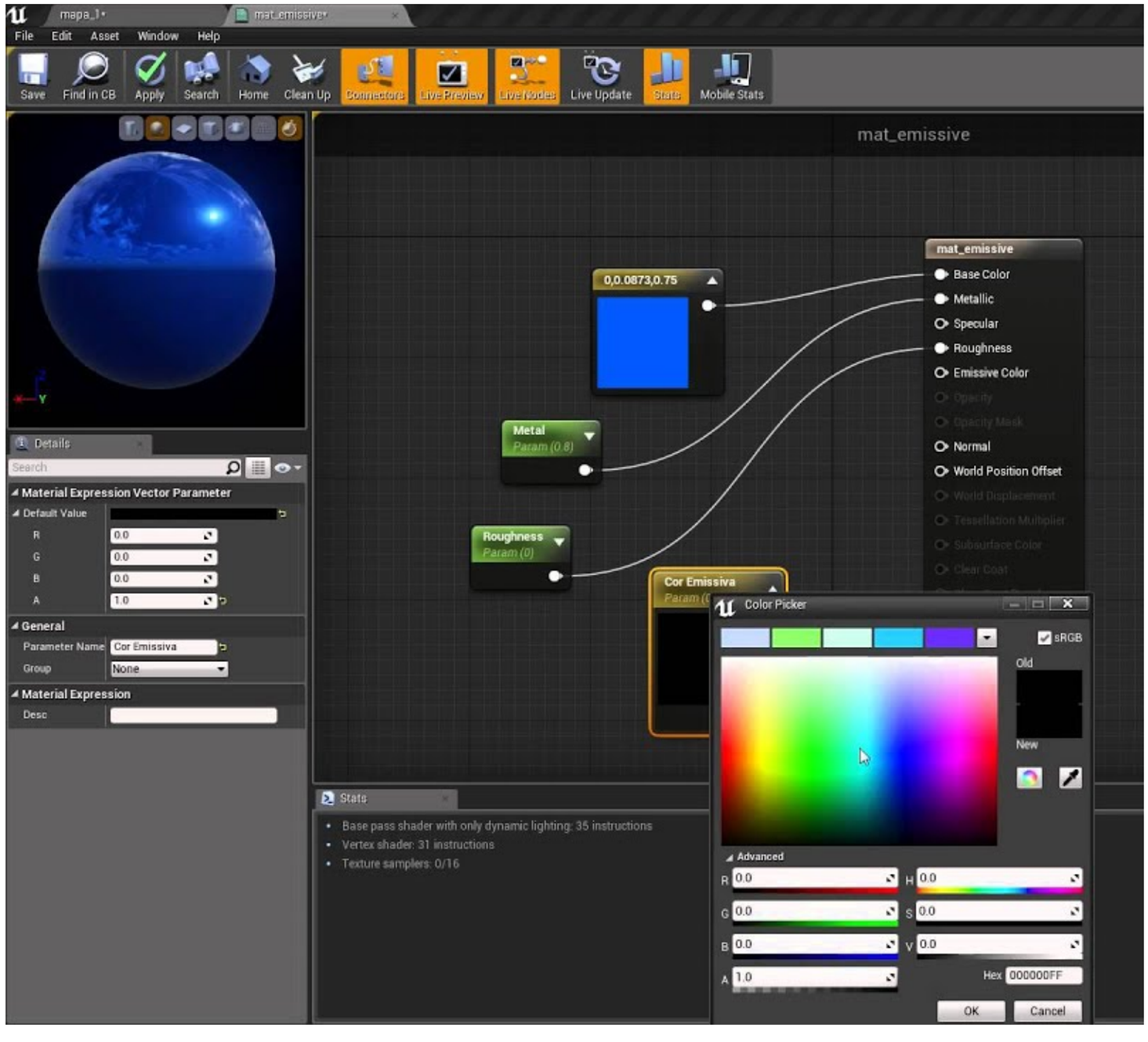


ARTIST TOOLS

- Tools related to the artist pipeline extend beyond the game engine
  - Maya/Max/Blender/ZBrush/Houdini for modeling
  - Substance/Houdini for procedural texture generation
  - Maya for animation
  - Houdini for VFX
- Game engine must provide ways to bring in this data, modify it for in-game use, and use it during gameplay

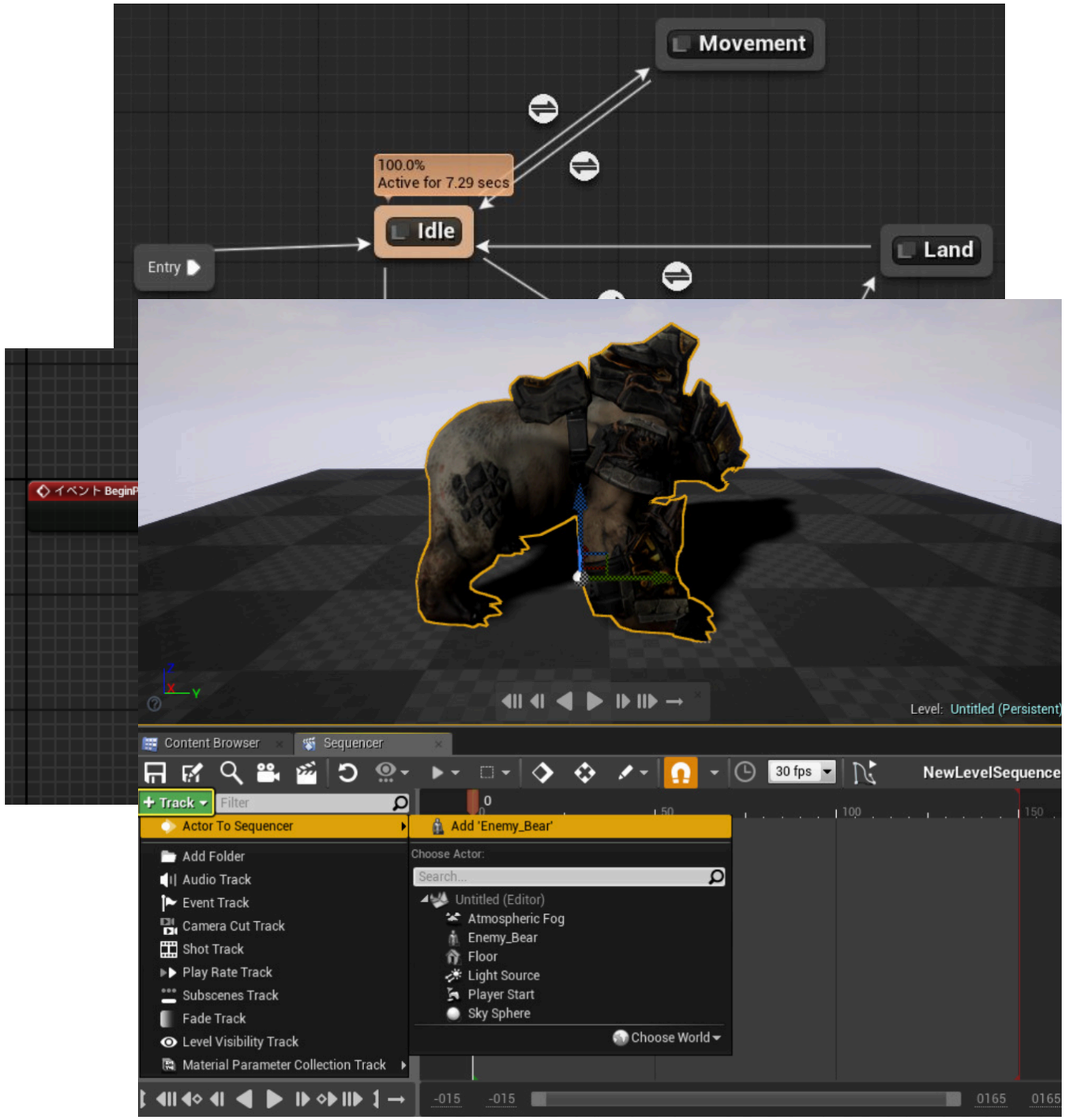
UE5 MATERIALS EDITOR

- Allows artists to create shaders in a node-based way
- Node-based material graphs standard practice in graphics pipeline
- Some tools for performance debugging



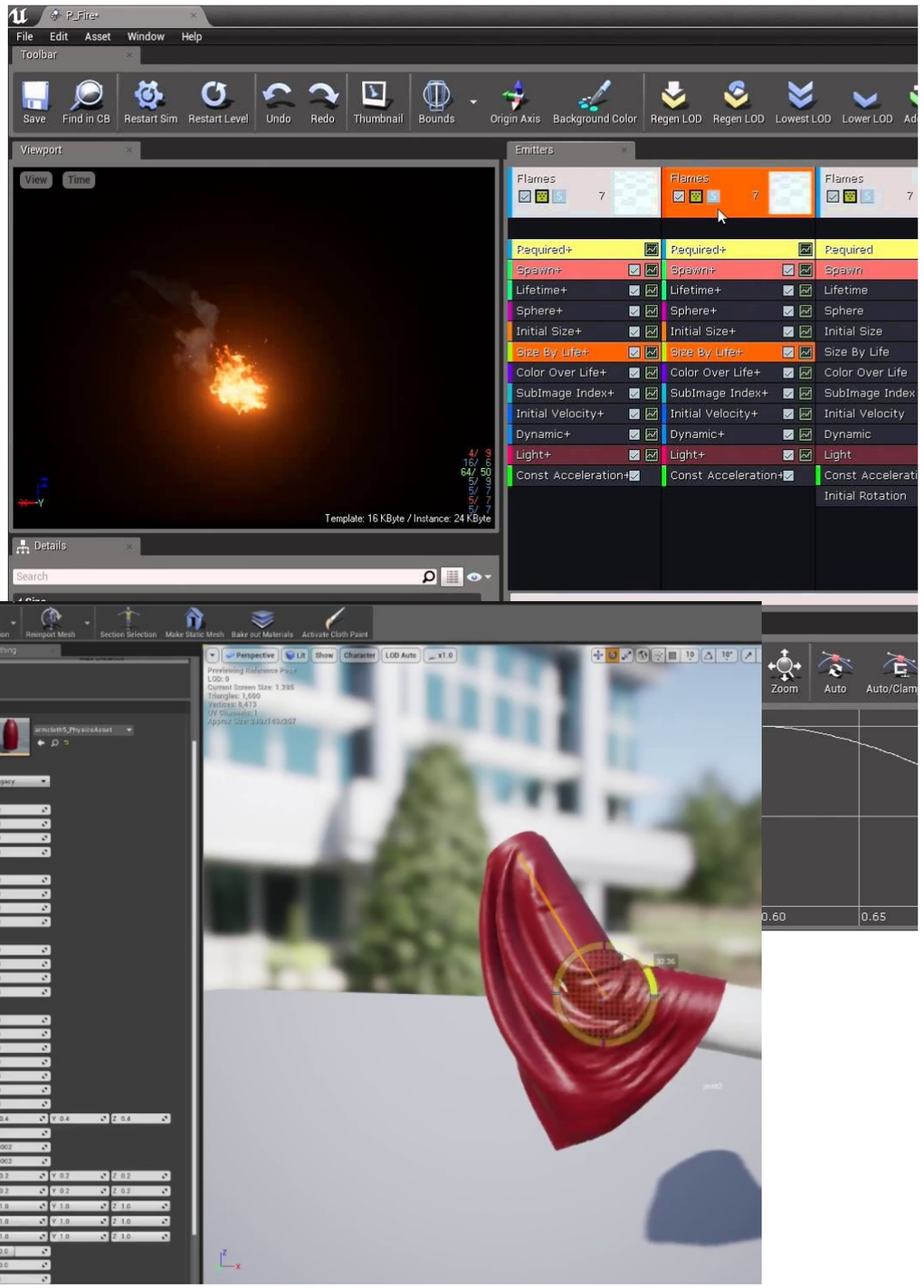
UE5 ANIMATION SYSTEMS

- Multiple systems to support skeletal, time-based, and cinematic animations
  - Animation Blueprints/ State Machines
  - Timelines
  - Sequencer



UE5 VFX SYSTEMS

- Multiple systems to support visual special effects
  - Particle systems
  - Hair and cloth simulation
  - Post-processing shaders
  - Material shaders



SUBSYSTEMS

- Run-time object model
- Real-time object model updating
- Messaging and event handling
- Scripting
- Level management and streaming

MEMORY MANAGEMENT

- Memory and performance are big considerations in game development
  - Nice-looking games need to run on consoles and phones at decent frame rates
- Engine design should facilitate performant code
  - Build for intelligent use of **garbage collection** and **smart pointers** to keep developer code clean and easy to reason about

HIGH-LEVEL INTERACTIONS

- Developers should work on as high a level as performance allows
  - Easier to reason about
  - Easier to structure
  - More reusable code
- Many game engines are written and optimized in C++
  - Support higher level scripting languages on top of this
  - Support visual scripting languages for artists and designers
- If your entire game is nothing but C++ (or equivalent low-level language), there may be a problem
  - We’re here to make games -- not programmer flex at each other

UE5'S STRUCTURE

- Designed to facilitate collaboration between programmers, artists, and designers

    1. Engine provides general functionality with an efficient implementation for most game features

    2. Game programmers create building-blocks for specific needs in **UE5-specific subset of C++**

    3. Designers and artists build on top of building blocks in node-based visual scripting language called **Blueprint**

- We will work primarily in C++ but also use Blueprint to better understand UE5's architecture and how to collaborate with designers/artists