# Week1
# Game Engine Development I

Hooman Salamat

# Instructor

Hooman Salamat (Lectures & Labs)

- Hooman.Salamat@georgebrown.ca
- Other contact info on Brightspace

# Assessment

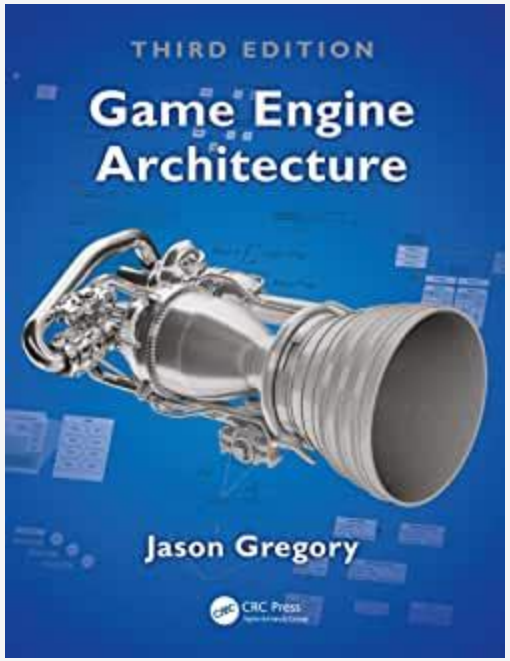
2x Assignments 20%

(10) Labs 20%

1x Midterm Exam 30%

1x Final Exam 30%

# Textbook




Game Engine Architecture, Third Edition

By: Jason Gregory

ISBN-13: 978-1-1380-3545-4

Publisher: CRC Press

# Objective & Concept

**Objective**

- This course provides students with an in-depth exploration of 3D game engine architecture.
- Students will learn state-of-the-art software architecture principles in the context of game engine design, investigate the subsystems typically found in a real production game engine, survey some engine architectures from actual shipping games, and explore how the differences between game genres can affect engine design.
- Students will participate in individual hands-on lab exercises, and also work together like a real game development team to design and build their own functional game engine by designing and implementing engine subsystems and integrating 3rd party components.

**Concepts**

- Engine subsystems including rendering, audio, collision, physics and game world models.
- Large-scale C++ software architecture in a games context.
- Tools pipelines for modern games.

# Course Materials

- Game Engine Development 1 - Course Materials: https://github.com/hsalamat/GAME3121

# Week 1 – Introduction

- - Course overview
- - What is a game engine?
- *game engines* like Epic Games' Unreal Engine, Valve's Source engine and, Crytek's CRYENGINE®, Electronic Arts DICE's Frostbite™ engine, and the Unity game engine—have become fully featured reusable software development kits that can be licensed and used to build almost any game imaginable.
- - Engine differences between game genres
- - Survey of runtime engine subsystems
  - Virtually all game engines contain a familiar set of core components, including the rendering engine, the collision and physics engine, the animation system, the audio system, the game world object model, the artificial intelligence system and so on.
- - Survey of tools and the asset pipeline
- **Reading:**
- - Course text: 1.2 – 1.7

# Structure of a game team

Game studios are usually composed of five basic disciplines:

- Engineers (design and implement the software that makes the game, and the tools, work)
  - *Runtime* programmers (who work on the engine and the game itself)
  - *tools* programmers (who work on the offline tools that allow the rest of the development team to work effectively).
- Artists:
  - "Content is king."
  - The artists produce all of the visual and audio content in the game, and the quality of their work can literally make or break a game.
  - *Concept artists, 3D modelers, Texture artists, Lighting artists, Animators, Motion capture actors, Sound designers, Voice actors, composers*
- Game Designers(design the interactive portion of the player's experience, typically known as *gameplay)*
  - Responsible for gameplay(story line, puzzles, levels, weapons)
  - Employ a game writer whose job can range from collaborating with the senior game designers to construct the story arc of the entire game
- Producers (manage the schedule and serve as a human resources manager, liaisons between the development team and the business unit of the company)
- Publisher (The marketing, manufacture and distribution of a game title)
  - A publisher is typically a large corporation, like Electronic Arts, THQ, Vivendi, Sony, Nintendo, etc.
  - Electronic Arts directly manages its studios. *First-party developers* are game studios owned directly by the console manufacturers (Sony, Nintendo and Microsoft). These studios produce games exclusively for the gaming hardware manufactured by their parent company.
- Other Staff (legal, information technology/technical support, administrative, etc.)

# What is a Game?

Raph Koster is the lead designer of Ultima Online and the creative director behind Star Wars Galaxies

*In his book, "A Theory of Fun for Game Design"*, Raph Koster defines a game:

Interactive experience that provides the player with an increasingly challenging sequence of patterns which he or she learns and eventually masters

This includes lots of things, but the core idea is that the "fun" is experience during the eureka moment

For the purposes of this course, we'll focus on the subset of games that comprise two- and three-dimensional virtual worlds with a small number of players (between one and 16 or thereabouts).

Much of what we'll learn can also be applied to HTML5/JavaScript games on the Internet, pure puzzle games like *Tetris*, or massively multiplayer online games (MMOG).

Our primary focus will be on game engines capable of producing first-person shooters, third-person action/platform games, racing games, fighting games and the like.

# Games for us

Focus primarily on 2D and 3D virtual worlds. Most of these can be described as *soft real-time agent-based computer simulations*.

*In soft real-time systems*—if the frame rate dies, the human player generally doesn't! Contrast this with a *hard real-time system*, in which a missed deadline could mean severe injury to or even the death of a human operator. The avionics system in a helicopter or the control-rod system in a nuclear power plant are examples of hard real-time systems.

In most video games, some subset of the real world—or an imaginary world using a mathematical model so that it can be manipulated by a computer

The model (*analytic* or *numerical)* is an approximation to and a simplification of reality: a *simulation* of the real or imagined game world.

Approximation and simplification are two of the game developer's most powerful tools.

In an *agent-based* simulation, a number of distinct entities known as "agents" interact.

They are interactive so have at least one actor or agent, where the agents are vehicles, characters, fireballs, power dots and so on.

Most games nowadays are implemented in an object oriented programming language

All interactive video games are "*temporal simulations*", meaning that the virtual game world model is *dynamic*—the state of the game world changes over time as the game's events and story unfold.

Most video games present their stories and respond to player input in real time, making them *interactive real-time simulations (exception turn-based games like chess)*.

They have some (loose) real-time constraints.

# What is a game engine?

•The term game engine arose in the 1990s.

•Doom by id Software was at the center (On June 24, 2009, ZeniMax Media acquired the company).

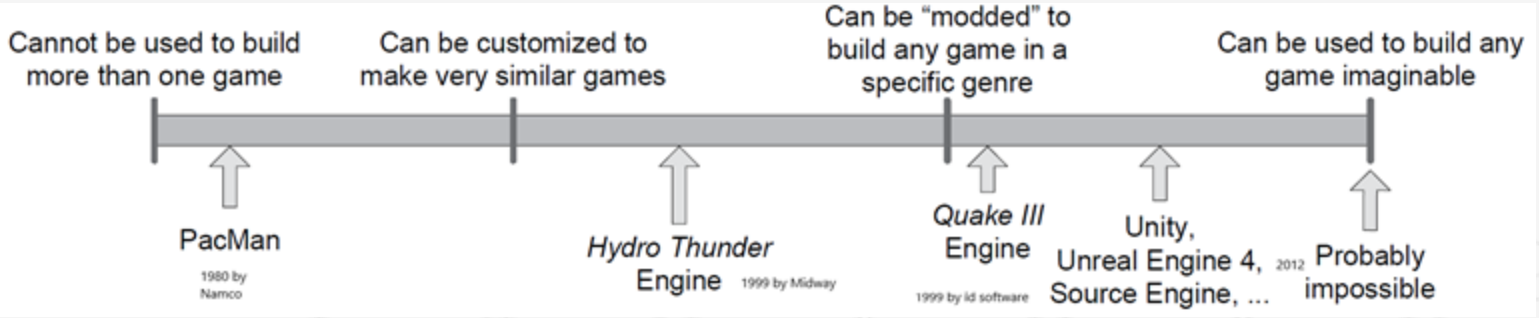
•The core components were separated from the game content

•Quake III and Unreal were designed with the separation in mind

- Sold licenses to their engine and tools
- Engine licensing began to be a viable secondary revenue stream for the developers who created them.
- You can license a game engine and reuse significant portions of its key software components in order to build games.
- Some of you may have done modding using these tools

•Game engine are data-driven architectures that are reusable and therefore do not contain game content – mostly true

•Game Engine Reusability Gamut



# *data-driven architecture*

a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine.

When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a different game.

“game engine” is a software that is extensible and can be used as the foundation for many different games without major modification.

The target hardware on which a game runs.

A rendering engine that was designed to handle intimate indoor environments probably won’t be very good at rendering vast outdoor environments.

The indoor engine might use a binary space partitioning (BSP) tree or portal system to ensure that no geometry is drawn that is being occluded by walls or objects that are closer to the camera.

In 1993, Doom was the first video game to make use of BSP after John Carmack utilized the most efficient 1991 algorithms describing front-to-back rendering with the use of a specialized data structure to record parts of the screen that had been drawn already.

The outdoor engine, on the other hand, might use a less-exact occlusion mechanism, or none at all, but it probably makes aggressive use of level-of-detail (LOD) techniques to ensure that distant objects are rendered with a minimum number of triangles, while using high-resolution triangle meshes for geometry that is close to the camera.

# First-person shooters

The first-person shooter (FPS) genre is typified by games like *Quake*, *Unreal Tournament*, *Half-Life*, *Battlefield*,

*Destiny*, *Titanfall* and *Overwatch*


FPS focuses on:

- On-foot locomotion, rail-confined or free roaming ground vehicles, hovercraft, boats and aircraft.
- Efficient rendering of large 3d worlds
- Responsive camera control
- High-fidelity animations and artificial intelligence for the non-player characters (NPCs)—the player's enemies and allies
- Cool weapons
- Forgiving physics model
- Small-scale online multiplayer capabilities (between 10 and 100 simultaneous players)
- The ubiquitous "death match" gameplay mode.

Rendering technology focuses on optimization for the environment

For example, indoor "dungeon crawl" games often employ binary space partitioning trees or portal-based rendering systems.

Outdoor FPS games use other kinds of rendering optimizations such as occlusion culling, or an offline sectorization of the game world with manual or automated specification of which target sectors are visible from each source sector.

# Platformers and the other third-person games

Third-person shooter(TPS) is a subgenre of 3D shooter games in which the player character is visible on-screen during gaming, and the gameplay consists primarily of shooting.

“Platformer” is the term applied to third-person character-based action games where jumping from platform to platform is the primary gameplay mechanic.

Typical games from the 2D era include *Space Panic*, *Donkey Kong*, *Pitfall!* And *Super Mario Brothers*.

The 3D era includes platformers like *Super Mario 64*, *Crash Bandicoot*, *Rayman 2*, *Sonic the Hedgehog*, the *Jak and Daxter* series, Ratchet and Clank, Gears of War

Platformers can usually be lumped together with third-person shooters

Third-person action/adventure games like *Just Cause 2*, *Gears of War 4,* the *Uncharted* series, the *Resident Evil* series, *The Last of Us* series, *Red Dead Redemption 2*, and the list goes on.

In a platformer, the main character is often cartoon-like and not particularly realistic or high-resolution.

Third-person shooters often feature a highly realistic humanoid player character.

Focus on

- Puzzle like elements
- Moving environmental objects
- Third person follow camera
- Complex camera collision system
- Main character’s abilities and locomotion modes

# Fighting games


Games like Tekken (Figure), Fight Night, and Soul Calibur

Technology focus on

- •Fighting animations
- •Hit detection
- •User input system capable of detecting complex button and joystick combinations
- •Crowds

Modern fighting games like EA's *Fight Night Round 4* and NetherRealm Studios' *Injustice 2* have upped the technological ante with features like:

· high-definition character graphics;

· realistic skin shaders with subsurface scattering and sweat effects;

· photo-realistic lighting and particle effects;

· high-fidelity character animations;

· physics-based cloth and hair simulations for the characters.

# Racing games

The genre has many subcategories:

Simulation-focused racing games ("sims") aim to provide a driving experience that is as realistic as possible (e.g., *Gran Turismo*).

Arcade racers favor over-the-top fun over realism (e.g., *San Francisco Rush*, *Cruis'n USA*, *Hydro Thunder*).

Street racing with tricked out consumer vehicles (e.g., *Need for Speed*, *Juiced*).

Kart racing is a subcategory in which popular characters from platformer games or cartoon characters from TV are re-cast as the drivers of whacky vehicles (e.g., *Mario Kart*, *Jak X*, *Freaky Flyers*).

Technology tricks include

- Using simple cards for background objects
- Track is broken down into sectors(2D regions)
- Third-person and first person cameras
- Camera collision with background geometry

# Real-time strategy


In this genre, the player deploys the battle units in his or her arsenal strategically across a large playing field

in an attempt to overwhelm his or her opponent.

The modern strategy game genre was arguably defined by *Dune II: The Building of a Dynasty* (1992).

Other games include *Warcraft*, *Command & Conquer*, *Age of Empires(figure)* and *Starcraft*.

Technology involves

- Each unit is relatively low-res, so that the game can support large numbers of them on-screen at once.
- Height-field terrain is usually the canvas upon which the game is designed and played.
- The player is often allowed to build new structures on the terrain in addition to deploying his or her forces.
- User interaction is typically via single-click and area-based selection of units, plus menus or toolbars containing commands, equipment, unit types, building types, etc.

# Massively Multiplayer Online Games (MMOG)

*Guild Wars 2* (AreaNet/NCsoft), *EverQuest* (989 Studios/SOE), *World of Warcraft* (Blizzard) and *Star Wars Galaxies* (SOE/Lucas Arts)

MMOG supports huge numbers of simultaneous players (from thousands to hundreds of thousands), usually all playing in one very large, *persistent* virtual world

Very powerful battery of servers

These servers maintain the authoritative state of the game world, manage users signing in and out of the game, provide inter-user chat or voice-over-IP (VoIP) services and more.

Graphics fidelity in an MMO is almost always lower than its non-massively multiplayer counterparts

Figure shows a screen from Bungie's FPS game, *Destiny*

*Destiny* has been called an MMOFPS because it incorporates some aspects of the MMO genre.

# Player-authored content

As social media takes off, games are becoming more and more collaborative in nature.

A recent trend in game design is toward *player-authored content*

For example, Media Molecule's *LittleBigPlanet™ 3: The Journey Home* are technically *puzzle platformers*, but they encourage players to create, publish and share their own game worlds.

Media Molecule's latest installment in this engaging genre is *Dreams* for the PlayStation 4

The most popular game today in the player-created content genre is *Minecraft*

# Virtual Reality

Computer-generated VR (CG VR) is a subset of this VR technology in which the virtual world is exclusively generated via computer graphics.

Headsets: HTC Vive, Oculus Rift, Sony PlayStation VR, Samsung Gear VR or Google Daydream View

*In augmented reality* (AR) and *mixed reality* (MR) a viewing device like a smart phone, tablet or tech-enhanced pair of glasses displays a real-time or static view of a real-world scene, and computer graphics are overlaid on top of this image.

"augmented reality" describes technologies in which computer graphics are overlaid on a live, direct or indirect view of the real world, but are not anchored to it.

"mixed reality" describes the use of computer graphics to render imaginary objects which are anchored to the real world and appear to exist within it.

Example of AR: US Army "tactical augmented reality" (TAR)

Example of MR: Microsoft's HoloLens, Android Camera App supports Google Pixel Playground (*AR Stickers)*

https://owlchemylabs.com/games/
Job simulator, Vacation Simulator

# VR Game Engines

- Similar in many respects to first-person shooter engines
- Many FPS-capable engines such as Unity and Unreal Engine support VR "out of the box."
- VR games differ from FPS games:
- *Stereoscopic rendering:* A VR game needs to render the scene twice, once for each eye.
- *Very high frame rate*. Studies have shown that VR running at below 90 frames per second is likely to induce disorientation, nausea, etc.
- *Navigation issues:* joypad or the WASD keys vs. the user physically walking around in the real world. Travelling by "flying" tends to induce nausea as well, so most games opt for a point-and click teleportation mechanism to move the virtual player/camera across larger distances.
- VR advantages:
- · users can reach in the real world to touch, pick up and throw objects in the virtual world;
- · a player can dodge an attack in the virtual world by dodging physically in the real world;
- · new user interface opportunities are possible, such as having floating menus attached to one's virtual hands, or seeing a game's credits written on a whiteboard in the virtual world;
- · a player can even pick up a pair of virtual VR goggles and place them onto his or her head, thereby transporting them into a "nested" VR world "VR-ception."

# Location-Based Entertainment

GAMES LIKE *POKÉMON GO* NEITHER OVERLAY GRAPHICS ONTO AN IMAGE OF THE REAL WORLD, NOR DO THEY GENERATE A COMPLETELY IMMERSIVE VIRTUAL WORLD.

THE GAME IS AWARE OF YOUR ACTUAL LOCATION IN THE REAL WORLD

PROMPTING YOU TO GO SEARCHING FOR POKÉMON IN NEARBY PARKS, MALLS AND RESTAURANTS

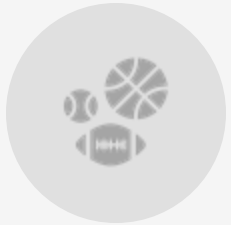
THIS KIND OF GAME CAN'T REALLY BE CALLED AR/MR/VR

SOME PEOPLE DESCRIBE IT AS LOCATION-BASED ENTERTAINMENT AND SOME AS AR

# Other Genres

- SPORTS, WITH SUBGENRES FOR EACH MAJOR SPORT (FOOTBALL, BASEBALL, SOCCER, GOLF, ETC.);

- ROLE-PLAYING GAMES (RPG);



- GOD GAMES, LIKE *POPULOUS* AND *BLACK & WHITE*;

- ENVIRONMENTAL/SOCIAL SIMULATION GAMES, LIKE *SIMCITY* OR *THE SIMS*;

- PUZZLE GAMES LIKE *TETRIS*;

- CONVERSIONS OF NON-ELECTRONIC GAMES, LIKE CHESS, CARD GAMES, GO, ETC.;

- WEB-BASED GAMES, SUCH AS THOSE OFFERED AT ELECTRONIC ARTS' POGO SITE;

# Some current engines

## The Quake Family of Engines

- The first 3D FPS game *Castle Wolfenstein 3D* (1992)
- *Doom*, *Quake*, *Quake II* and *Quake III*
- All of these engines are very similar in architecture
- Has lineage that extends to modern games like *Medal of Honor*
- *Quake* and *Quake II* engines source code are freely available (https://github.com/id-Software/Quake-2)

## Unreal Engine

- Epic Games, Inc. burst onto the FPS
- Pros:Industry leader for photorealism (Nanite virtualized geometry, Lumen global illumination).Massive ecosystem, AAA support, free until revenue threshold.Widely used for games, film, architectural visualization, VR.
- Cons:Heavy system requirements.Steeper learning curve for beginners.
- Examples: Fortnite, The Matrix Awakens demo, Hellblade II.
- Best for: AAA games, cinematic visuals, virtual production.

## Unity

- Pros:Extremely versatile (2D, 3D, VR, mobile, indie).HDRP (High-Definition Render Pipeline) for high-end graphics.URP (Universal Render Pipeline) for cross-platform performance.Large asset store, strong developer community.
- Cons:HDRP visuals not as cutting-edge as Unreal 5.Complex pipeline setup for advanced graphics.
- Examples: Genshin Impact, Hollow Knight: Silksong, Cities: Skylines II.
- Best for: Indie games, mobile/VR, cross-platform development.

# More engines

## Godot Engine

- Pros:Open-source, completely free (MIT license).Lightweight, great for 2D and mid-tier 3D.Active development, growing community.
- Cons:3D rendering still catching up with Unreal/Unity.Fewer AAA titles made with it.
- Examples: Cassette Beasts, Deep Rock Galactic: Survivor.
- Best for: Indies, 2D/3D hybrid projects, developers who want full control.

## Source Engine

- Games like *Half-life 2* and its sequels, *Team Fortress 2,* and *Portal*
- Very powerful with good graphics capabilities and a good toolset

## DICE's Frostbite by EA

- Pros:Proprietary engine with strong graphics (used in Battlefield).Optimized for realism and large-scale environments.
- Cons:Not publicly available—only EA studios use it.
- Examples: Battlefield series, FIFA / EA Sports FC, Mass Effect Andromeda.
- Best for: AAA realism (internal EA projects).

## Rockstar Advanced Game Engine (RAGE)

- Pros:Highly optimized for open-world realism.Used in some of the most visually stunning games.Cons:Proprietary, not available outside Rockstar.
- Examples: GTA V, Red Dead Redemption 2.
- Best for: Open-world AAA realism (internal Rockstar).

# Even more engines

## CryEngine by Crytek

- Pros:Known for realistic lighting and vegetation rendering.Powerful out-of-the-box visuals.Royalty-based model.
- Cons:Smaller ecosystem than Unreal/Unity.Steep learning curve, limited tutorials.
- Examples: Crysis series, Hunt: Showdown.
- Best for: Realistic shooters, open-world games.

## Sony's PhyreEngine

- Uses to create games for the Sony platforms
- Numerous titles have been written with this engine: *flOw*, *Flower* and *Journey*, and Coldwood Interactive's *Unravel*
- It is available free of charge to any licensed Sony developer as part of the PlayStation SDK

## Amazon Lumberyard / Open 3D Engine (O3DE)

- Pros:Open-source (Apache 2.0).Based on CryEngine, but expanded.Backed by AWS integration (cloud, multiplayer).
- Cons:Still maturing, less polished than Unreal/Unity.Smaller adoption.
- Examples: Star Citizen (heavily customized Lumberyard).
- Best for: Studios wanting open-source flexibility + cloud integration.

# Open Source Engines

Gnu Public License (GPL) : code to be freely used by anyone, as long as their code is also freely available

Lesser Gnu Public License (LGPL): code to be used even in proprietary for-profit applications

http://en.wikipedia.org/wiki/List_of_game_engines

OGRE is a well-architected, easy-to-learn and easy-to-use 3D rendering engine.

- https://www.ogre3d.org/
- advanced lighting and shadows
- a good skeletal character animation system
- a two-dimensional overlay system for heads-up displays and graphical user interfaces
- A post-processing system for full-screen effects like bloom

Some other well-known open source engines: Panda3D, Yake, Crystal Space, Torque and Irrlicht

The Amazon Lumberyard / Open 3D Engine (O3DE) engine does provide source code to its developers. It is a free cross-platform engine developed by Amazon, and based on the CRYENGINE architecture.

# Recommendations

Photorealistic AAA → Unreal Engine 5.

Indie / Mobile / Cross-platform → Unity or Godot.

Realistic FPS / Open-world → CryEngine or O3DE.

Custom AAA studios → Frostbite (EA), RAGE (Rockstar).

# 2D Engines

Designed for non-programmers to build apps for Android and iPhone

*Multimedia Fusion 2* (http://www.clickteam.com) is a 2D game/multimedia authoring toolkit developed by ClickteamGame

The Games Factory 2 (https://www.clickteam.com/the-games-factory-2), are also used by educational camps like PlanetBravo (http://www.planetbravo.com)

- to teach kids about game development and programming/logic concepts
- Fusion supports the iOS, Android, Flash, and Java platforms.

*Game Salad Creator* (http://gamesalad.com/creator) is another graphical game/multimedia authoring toolkit aimed at non-programmers

*Scratch* (http://scratch.mit.edu) is an authoring toolkit and graphical programming language that can be used to create interactive demos and simple games.

# Runtime Engine Architecture

Consists of the tools suite and runtime components

Figure shows all of the major runtime components that make up a typical 3D game engine

Game engines are definitely large software systems

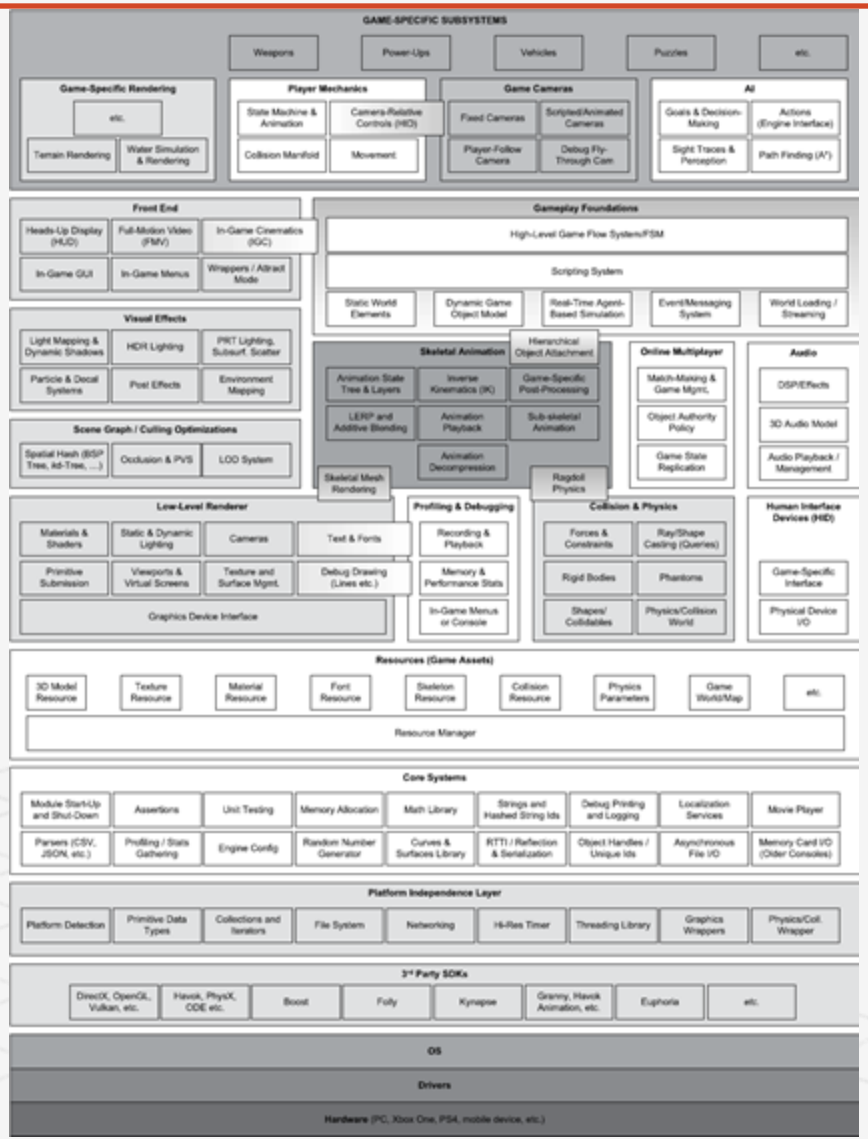
This diagram doesn't even account for all the tools.

Game engines are built in *layers*.

Upper layers depend on lower layers

When a lower layer depends upon a higher layer, we call this a *circular dependency*.

Avoids circular dependencies to maximize reuse and testability



# Runtime Engine

Low level components

3rd Party SDKs

Platform independence layer

Core systems

Resources manager

Rendering engine

Profiling/Debugging

Collisions and Physics

Animation

Human Interface Devices

Audio

Gameplay foundation system

AI

# Low level components


Target Hardware

This is the system (or platforms) or console that the game is to run on

Device Drivers

Device drivers are low-level software components provided by the operating system or hardware vendor

Shield the OS and upper layers from low level device communications details

Operating System

Handles the execution and interruption of multiple programs on a single machine (preemptive multitasking)

A PC game can never assume it has full control of the hardware

Very thin OS on an early consoles.

On Modern Consoles, The operating systems on the Xbox and PlayStation can interrupt the execution of your game, or take over certain system resources, in order to display online messages, or to allow the player to pause the game and bring up the PS4's "XMB" user interface or the Xbox One's dashboard, for example.

# 3rd Party SDKs


Data Structure and Algorithms

Games depend heavily on *container* data structures and algorithms to manipulate them:

Boost: a powerful data structures and algorithms library, designed in the style of the standard C++ library, Folly: library used at Facebook, and Loki: a powerful generic programming template library

STL – C++ standard template library data structures, strings, stream-based I/O (ex: generic container classes such as std::vector and std::list)

Graphics

Glide, *libgcm,* OpenGL, DirectX, *Edge, Vulkan, Metal*

Collisions and Physics

SDK for collision detection and rigid body dynamics: Havok, PhysX(NVIDIA), Open Dynamic Engine(ODE)

Character Animation

Granny from Rad Game Tools, Havoc Animation, OrbisAnim by SN Systems

Artificial Intelligence

Kythera AI, Havoc AI, RAIN AI, OpenAI Gym / Petting Zoo, TensorFlow/PyTorch, Recast & Detour

Biomechanical Character Models: *Endorphin and Euphoria (*produce character motion using advanced biomechanical models) produce character motion using advanced biomechanical models

# Platform independence layer

Allows the engine to be developed without the concern of the underlying platform


Provides wrappers to common target specific operations

Include things like primitive types, network, file systems, etc.

Companies like Electronic Arts and Activision Blizzard target their games at a wide variety of platforms

This layer sits atop the hardware, drivers, operating system and other third-party software

There are two primary reasons to “wrap” functions as part of your game engine’s platform independence layer

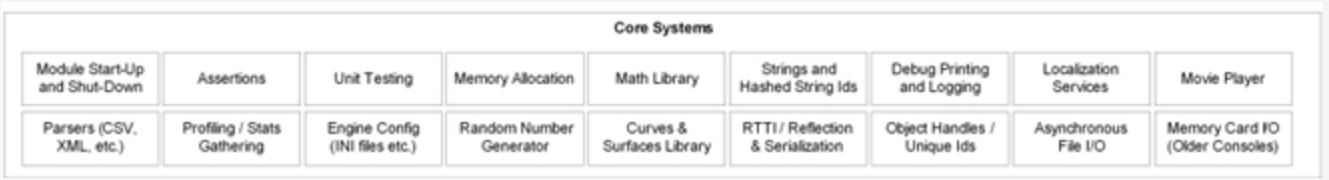
1. Some APIs like those provided by the operating system, or even some functions in older “standard” libraries like the C standard library differ significantly from platform to platform; wrapping these functions provides the rest of your engine with a consistent API across all of your targeted platforms.

2. Even when using a fully cross-platform library such as Havok, you might want to insulate yourself from future changes, such as transitioning your engine to a different collision/physics library in the future.

# Core systems

Every game engine requires a grab bag of useful software utilities.

We'll categorize these under the label "core systems."




*Assertions:* lines of error-checking code that are inserted to catch logical mistakes(stripped out of the final production build of the game)

Memory Management: every game engine implements its own custom memory allocation system

Math library: These libraries provide facilities for vector and matrix math, quaternion rotations, trigonometry, geometric operations, etc.

Custom data structures: any other data structures for managing fundamental data structures (linked lists, dynamic arrays, binary trees, hash maps, etc.) and algorithms (search, sort, etc.) -- data structures and algorithms other than Boost and Folly

# Resource manager

Provides a unified interface for accessing assets

The level of complexity is dictated by need


Some engines do this in a highly centralized and consistent manner (e.g., Unreal's packages, OGRE's Resource-Manager class).

Other engines take an ad hoc approach, often leaving it up to the game programmer to directly access raw files on disk or within compressed archives such as Quake's PAK files.

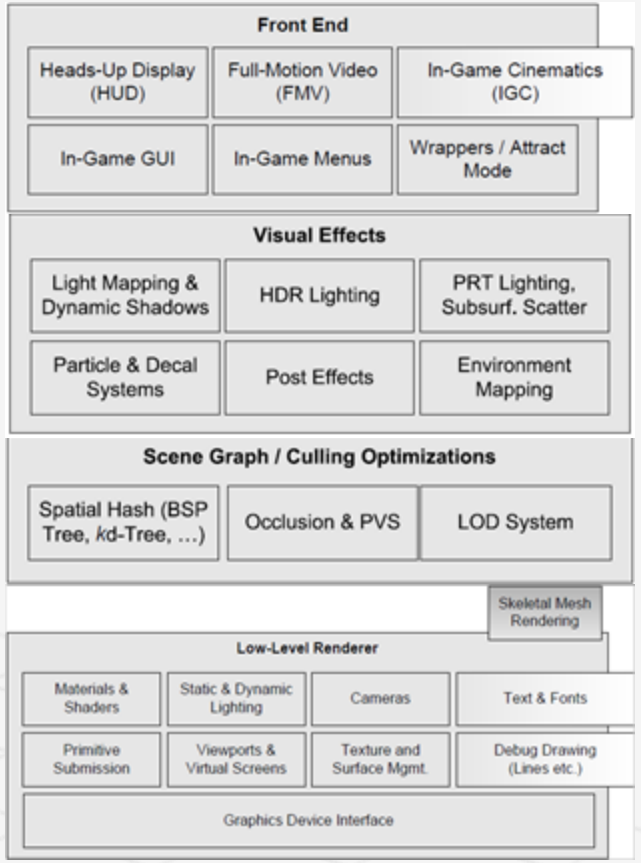
# Rendering engine

The rendering engine is one of the largest and most complex components of any game engine.

Most modern rendering engines share some fundamental design philosophies, driven in large part by the design of the 3D graphics hardware upon which they depend.

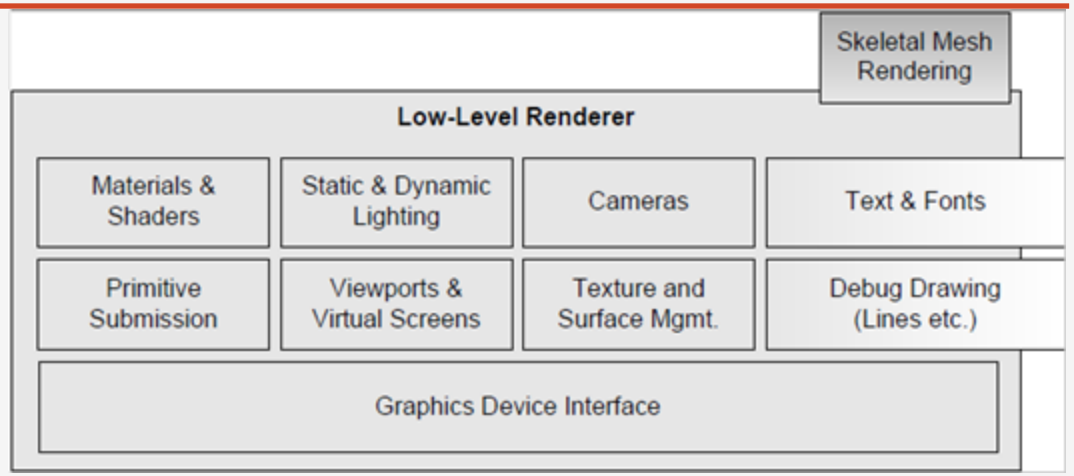
One common approach to rendering engine design is to employ a layered architecture

- Low-level Renderer
- Scene graph management
- Visual effects
- Front end



# Low-level Renderer

Encompasses all of the raw rendering facilities of the engine.




Focuses on rendering primitives as quickly and richly as possible

Does not consider visibility

It contains the following subcomponents:

Graphics Device Interface component:

Graphics SDKs, such as DirectX, OpenGL or Vulkan, require:

Access and enumerate the graphics devices
Initialize the GD
Setup buffering (back-buffer, stencil buffer)
"Message Pump": For a PC game engine, you also need code to integrate your renderer with the Windows message loop.

Other subcomponents:

Collect submissions of geometric primitives (render packets), such as meshes, line lists, point lists, particles, terrain patches, text strings

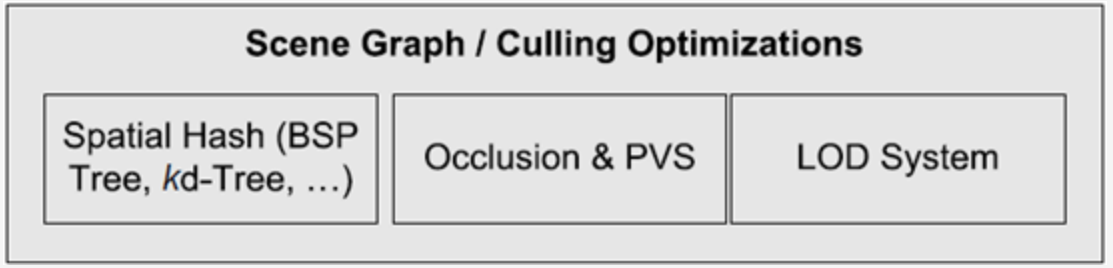
Provide abstraction of the camera interface (field of view, near/far plane locations

Material system: Each submitted primitive is associated with a material

Dynamic lighting system: Each submitted primitive is affected by *n* dynamic lights

Text and fonts

# Scene graph




The low-level renderer draws all of the geometry submitted to it, without much regard for whether or not that geometry is actually visible (other than back-face culling and clipping triangles to the camera frustum).

A higher-level component is usually needed in order to limit the number of primitives submitted for rendering, based on some form of visibility determination.

For very small game worlds, a simple *frustum cull* (i.e., removing objects that the camera cannot "see") is enough.

For larger game worlds, a more advanced *spatial subdivision* data structure might be used.

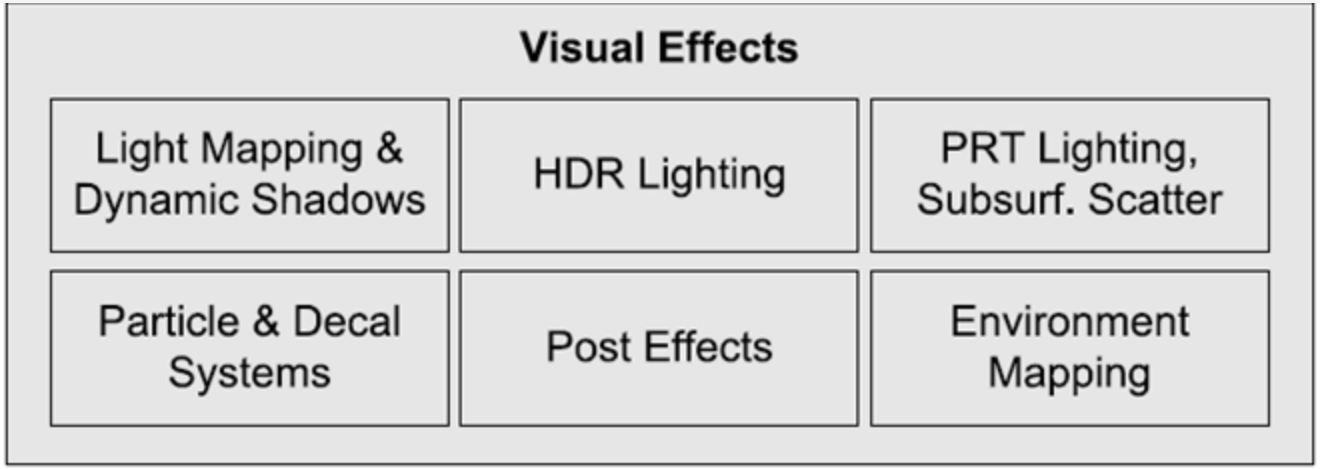
- allowing the potentially visible set (PVS) of objects to be determined
- Spatial subdivisions can take many forms, including a binary space partitioning tree, a quadtree, an octree, a *k*dtree or a sphere hierarchy.
- A spatial subdivision is sometimes called a scene graph
- Portals or occlusion culling methods might also be applied in this layer

The low-level renderer should be completely agnostic to the type of spatial subdivision or scene graph being used

OGRE provides a plug-and-play scene graph architecture.

# Visual effects

Modern game engines support a wide range of visual effects:




Particle systems (for smoke, fire, water splashes, etc.)

Decal systems (for bullet holes, foot prints, etc.)

Light mapping and environment mapping

Dynamic shadows

Full-screen post effects, applied after the 3D scene has been rendered to an off-screen buffer.

- high dynamic range (HDR) tone mapping and bloom;
- full-screen anti-aliasing (FSAA);
- color correction and color-shift effects, including bleach bypass, saturation and desaturation effects, etc.

The particle and decal systems are usually distinct components of the rendering engine and act as inputs to the low-level renderer.

Light mapping, environment mapping and shadows are usually handled internally within the rendering engine proper

Full-screen post effects are either implemented as an integral part of the renderer or as a separate component that operates on the renderer's output buffers.

# Front end

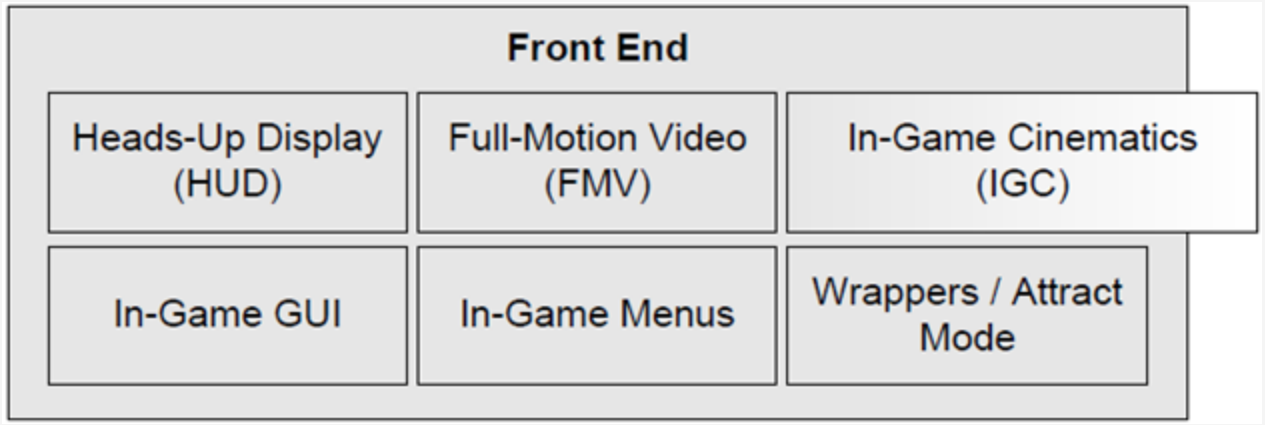
Most games employ some kind of 2D graphics overlaid on the 3D scene

for various purposes.

the game's *heads-up display* (HUD);

• in-game menus, a console and/or other *development tools*,

which may or may not be shipped with the final product;




• possibly an in-game *graphical user interface* (GUI), allowing the player to manipulate his or her character's inventory, configure units for battle or perform other complex in-game tasks.

Two-dimensional graphics are usually implemented by drawing textured quads (pairs of triangles) with an orthographic projection. Or they may be rendered in full 3D, with the quads bill-boarded so they always face the camera.

FMV plays full-screen movies that have been recorded earlier

IGC allows cinematic sequences to be choreographed within the game itself, in full 3D (ex: player walks through a city, a conversation between two key characters might be implemented as an in-game cinematic.)

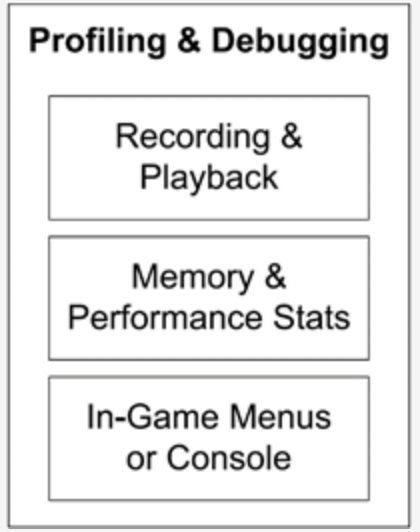
*Uncharted 4: A Thief's End*, have moved away from pre-rendered movies entirely, and display *all* cinematic moments in the game as real-time IGCs.

# Profiling/Debugging

Tools to profile the performance of their games in order to optimize performance.

Memory resources are usually scarce, so developers make heavy use of memory analysis tools

In-game debugging facilities, such as debug drawing, an in-game menu system or console and the ability to record and play back gameplay for testing and debugging purposes.




general-purpose software profiling tools: Intel's *Vtune,* IBM's *Quantify* and *Purify* (part of the *PurifyPlus* tool suite), *Insure++* by Parasoft, and *Valgrind* by Julian Seward and the Valgrind development team.

Most game engines also incorporate a suite of custom profiling and debugging tools:

- Code timing
- Display the profiling statistics on-screen while the game is running
- Dumping performance stats to a text file
- Determining memory usage used by the engine, and by each subsystem, including various on-screen displays
- Dumping memory usage, high water mark and leakage stats when the game terminates and/or during gameplay
- Record and playback game events for tracking down bugs.
- Print statement output control, along with an ability to turn on or off different categories of debug

# Collisions and Physics

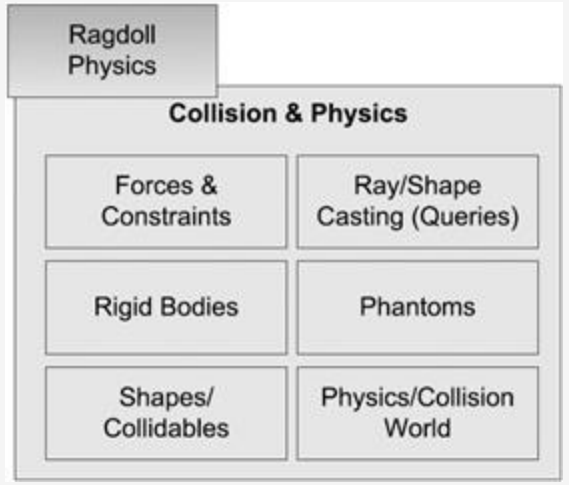
"physics system" in the game industry = rigid body dynamics

Focus: the motion (kinematics) of rigid bodies and the forces and torques (dynamics) that cause this motion to occur.

Collision and physics are usually quite tightly coupled.

Physics engine creation is its own unique undertaking

Most game companies use a third party SDK such as:




*Havok:* the gold standard in the industry today

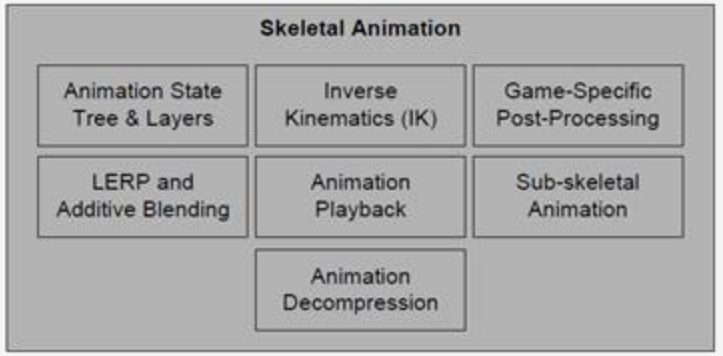
PhysX: *PhysX* by NVIDIA is another excellent collision and dynamics engine. It was integrated into Unreal Engine 4 and is also available for free as a stand-alone product for PC game development.

Open Dynamics Engine (ODE): An open source physics and collision engine (http://www.ode.org)

I-Collide, V-Collide and RAPID: non-commercial collision detection engines developed at (UNC)

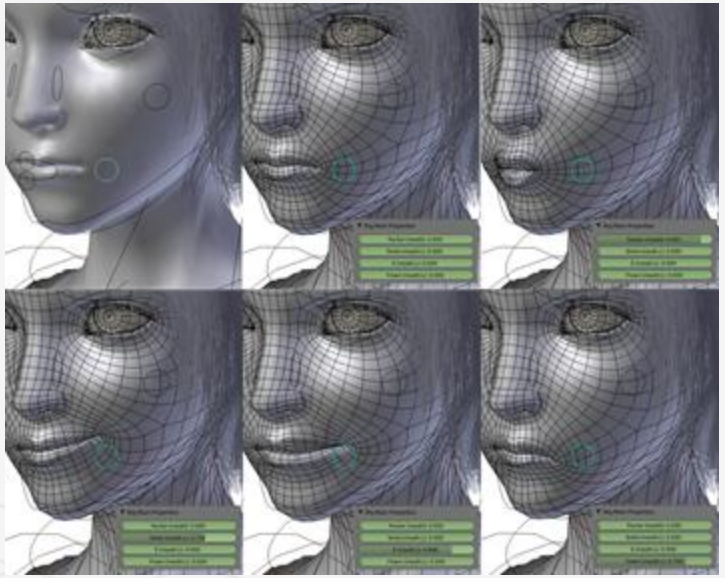
# Animation

Any game that has organic or semi-organic characters (humans, animals, cartoon characters or even robots) needs an animation system.




Five types of animation are used

- Sprite/texture animation
- Rigid body hierarchy animation
- Skeletal animation
- Vertex animation
- Morphing (morph target)



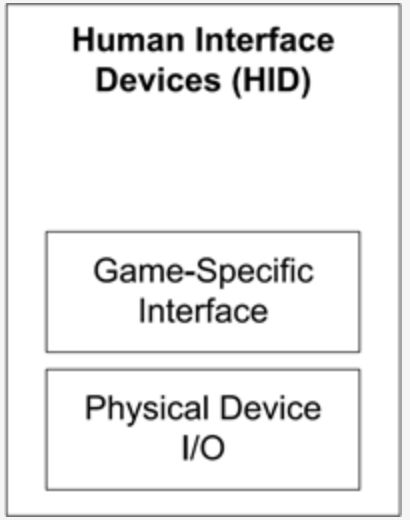
Skeletal animations still the most popular:

- The animation system produces a pose for every bone in the skeleton
- These poses are passed to the rendering engine as a palette of matrices.
- The renderer transforms each vertex by the matrix or matrices in the palette, in order to generate a final blended vertex position.
- This process is known as *skinning*.

# HID

Every game needs to process input from the player, obtained from various *human interface devices* (HIDs) including

- the keyboard and mouse,
- a joypad, or
- other specialized game controllers, like steering wheels, fishing rods, dance pads, the Wiimote, etc.




The HID engine component is sometimes architected to divorce the low level details of the game controller(s) on a particular hardware platform from the high-level game controls.

It massages the raw data coming from the hardware, introducing a dead zone around the center point of each joypad stick, debouncing button-press inputs, detecting button-down and button-up events, interpreting and smoothing accelerometer inputs (e.g., from the PlayStation Dualshock controller) and more.

# Audio

Often overlooked until the end

Varies in sophistication based on need

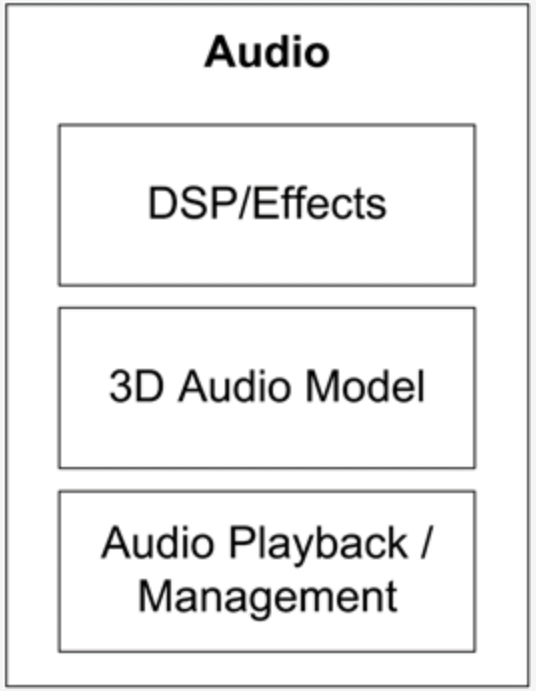
Quake's audio engine is pretty basic, and game teams usually augment it with custom functionality.

Unreal Engine 4 provides a reasonably robust 3D audio rendering engine.

For DirectX platforms (PC, Xbox 360, Xbox One), Microsoft provides an excellent runtime audio engine called XAudio2.

Electronic Arts has developed an advanced, high-powered audio engine internally called SoundR!OT

Sony Interactive Entertainment (SIE) provides a powerful 3D audio engine called Scream.



# Multiplayer/networking

Multiplayer games come in at least four basic flavors:

Single screen – multiple players on the same screen (*Smash Brothers*, *Lego, Star Wars* and *Gauntlet*)

Split-screen multiplayer – multiple perspectives on the same screen (multiple HIDs attached to a single game machine)

Networked multiplayer – multiple computers networked together

Massive multiplayer online games (MMOG) – hundreds of thousands of users can be playing simultaneously within a giant, persistent, online virtual world hosted by a powerful battery of central servers.


Multiplayer games are quite similar in many ways to their single-player counterparts.

Difficult to convert single to multiplayer, easy to do the opposite.

Many game engines treat single-player mode as a special case of a multiplayer game.

The Quake engine is well known for its *client-on-top-of-server* mode, in which a single executable, running on a single PC, acts both as the client and the server in single-player campaigns.

# Gameplay foundation system

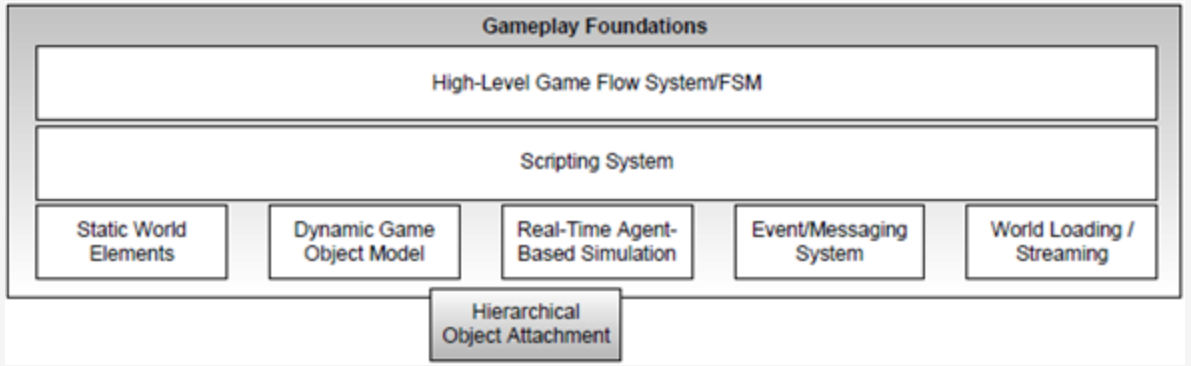
The term *gameplay* refers to the action that takes place in the game,

the rules that govern the virtual world in which the game takes place,

the abilities of the player character(s) (known as *player mechanics*)

and of the other characters and objects in the world,

and the goals and objectives of the player(s).




Gameplay is typically implemented either in the native language in which the rest of the engine is written or in a high-level scripting language—or sometimes both.

To bridge the gap between the gameplay code and the low-level engine systems that we've discussed thus far, most game engines introduce a layer that I'll call the *gameplay foundations* layer.

Gameplay foundations layer introduces a game world (both static and dynamic elements)

Game World are modelled in an object-oriented matter.

The collection of object types that make up a game is called the *game object model*

# Game Object Model

The game object model provides a real-time simulation of a heterogeneous collection of objects in the virtual game world:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| · static background geometry, like buildings, roads, terrain (often a special case), etc.; | · dynamic rigid bodies, such as rocks, soda cans, chairs, etc.; | · player characters (PC); | · non-player characters (NPC); | · weapons; | · projectiles; | · vehicles; | · lights (which may be present in the dynamic scene at runtime, or only used for static lighting offline); | · cameras; |

The game world model is intimately tied to a *software object model*

# Event system

**Objects need to communicate with one another.**

**For example, the object sending the message might simply call a member function of the receiver object.**

**An event driven architecture is also a common approach to inter-object communication.**

**In an event-driven system:**

The sender creates a little data structure called an *event* or *message*, containing the message's type and any argument data that are to be sent.

The event is passed to the receiver object by calling its *event handler function*.

Events can also be stored in a queue for handling at some future time.

# Scripting system

Many game engines employ a scripting language in order to make development of game-specific gameplay rules and content easier and more rapid.

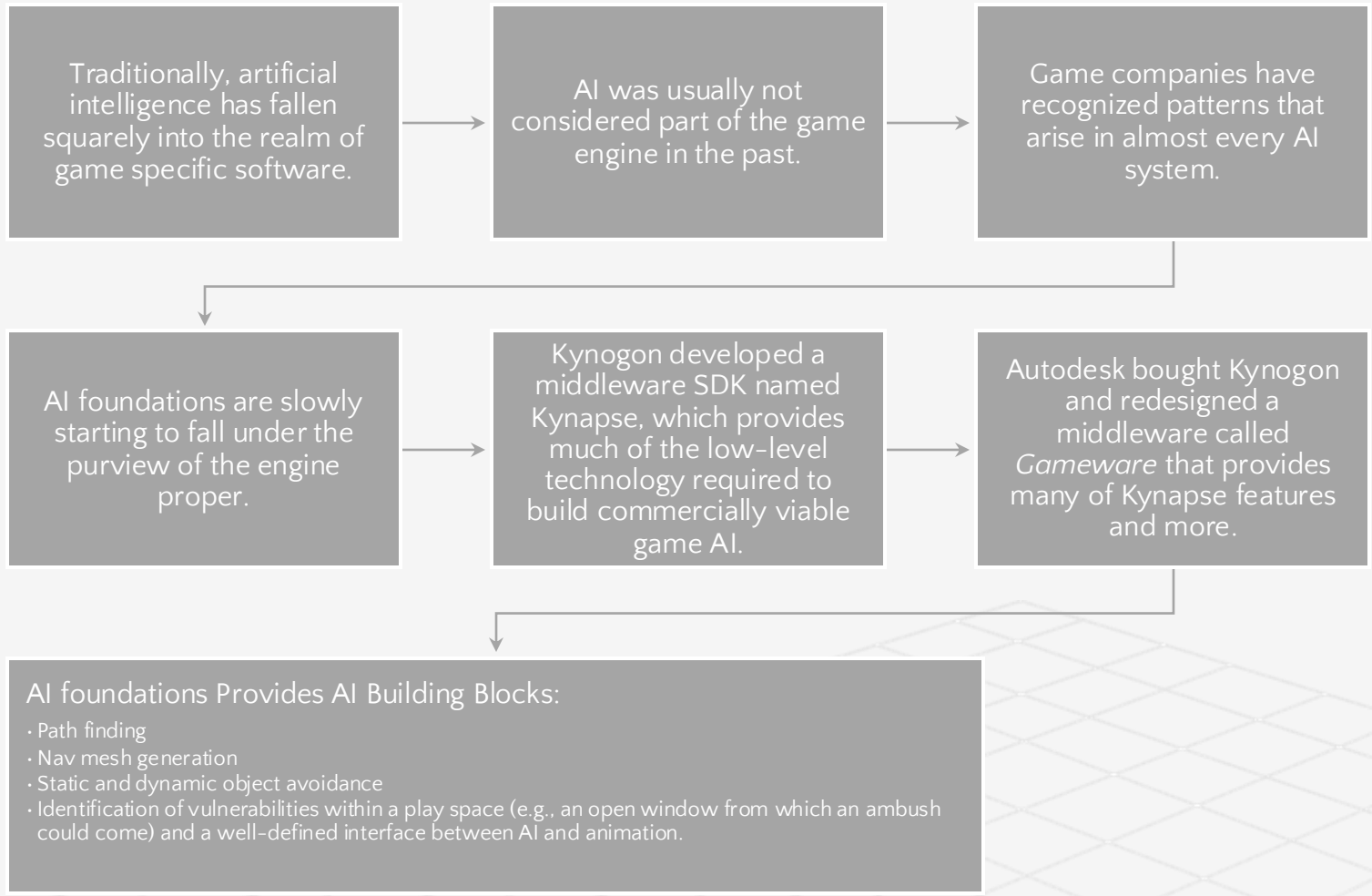
Allows for the creation on new game logic without recompiling

When a scripting language is integrated into your engine, changes to game logic and data can be made by modifying and reloading the script code.

Some engines allow script to be reloaded while the game continues to run.

Speeds software development considerably.

# Artificial intelligence foundations



# Built-in Game Engine AI Systems

Many popular game engines already come with strong AI tools:

1. **Unity ML-Agents**
   - Unity's open-source toolkit for training NPCs with reinforcement learning, imitation learning, and other ML techniques.
   - Great for research and commercial games.
2. **Unreal Engine (Behavior Trees + EQS)**
   - Built-in **Behavior Tree** system for decision-making and **Environment Query System (EQS)** for spatial reasoning.
   - Used in AAA titles (e.g., Fortnite).

# AI Middleware / Frameworks

These are specialized AI systems you can integrate into engines:

**Kythera AI**

- Middleware for Unreal and custom engines.
- Handles navigation meshes, behavior trees, and crowd simulation.

**Havok AI**

- Part of Havok's middleware suite (owned by Microsoft).
- Optimized for large-scale pathfinding and navigation.

**RAIN AI (for Unity)**

- Free AI toolkit (though not actively updated).
- Offers pathfinding, decision-making, and sensors.

# Machine Learning / Research-Oriented AI

For developers who want to push boundaries with ML-driven gameplay:

**TensorFlow / PyTorch (integrated with Unity or Unreal)**

- Often used with Unity ML-Agents or custom pipelines.
- Great for adaptive NPCs or AI opponents.

**OpenAI Gym / PettingZoo**

- For prototyping reinforcement learning in game-like environments.
- More for experimentation than production.

# Navigation & Pathfinding Libraries

Recast & Detour

- Industry-standard for navmesh generation and pathfinding.
- Used in many engines (Unity, Unreal, custom engines).

*A Pathfinding Project (Unity)**

- Popular Unity plugin for grid, navmesh, and point graph pathfinding.

# Classic AI Techniques (Engine-Agnostic)

Even without middleware, most studios rely on tried-and-true approaches:

**Finite State Machines (FSMs)** – simple, predictable AI.

**Behavior Trees** – modular, hierarchical logic for NPCs.

**GOAP (Goal-Oriented Action Planning)** – flexible planning AI (used in *F.E.A.R.*).

**Utility AI** – scoring-based decision-making (used in *The Sims 4*).

# Game AI Engines & Frameworks – Comparison

| AI Engine / Framework | Pros ✅ | Cons ❌ | Best Use Cases 🎯 | Skill Level |
|---|---|---|---|---|
| **Unity ML-Agents** | - Easy Unity integration - Supports reinforcement & imitation learning - Active community | - Training requires compute power - Can be overkill for simple NPCs | Smart adaptive NPCs, training bots, research experiments | Intermediate–Advanced |
| **Unreal Engine (Behavior Trees + EQS)** | - AAA-grade tools - Integrated with navmesh & animation - Powerful EQS for decision-making | - Steep learning curve - Less suited for ML-based AI out of the box | AAA shooters, RPGs, open-world games | Intermediate |
| **Kythera AI** | - Advanced navmesh & crowd simulation - Middleware for Unreal/custom engines - Handles large-scale environments | - Commercial license (not free) - Best for studios with budget | Large open worlds, MMO AI, simulation-heavy games | Advanced (Studio-level) |
| **Havok AI** | - Industry-proven - Highly optimized navigation/pathfinding - Supports dynamic worlds | - Proprietary & costly - Limited flexibility vs. custom ML | AAA titles, physics-heavy games needing scalable AI | Advanced |
| **RAIN AI (Unity)** | - Free Unity toolkit - Provides FSMs, sensors, decision-making - Easy setup | - No longer actively updated - Limited vs. ML-Agents | Simple NPC AI in Unity projects | Beginner–Intermediate |
| **Recast & Detour** | - Gold standard navmesh generator - Used in many commercial engines - Fast, stable | - Only handles navigation/pathfinding - No decision-making AI | Custom engines, Unity/Unreal extensions | Intermediate |
| A Pathfinding Project (Unity)* | - Very popular Unity asset - Grid, navmesh, & point graph support - Optimized for 2D & 3D | - Navigation only (no high-level AI) - Limited ML support | Indie projects, 2D/3D games with AI agents | Beginner–Intermediate |
| **TensorFlow / PyTorch (with Unity/Unreal)** | - Cutting-edge ML libraries - Great for research & adaptive AI - Integrates with Unity ML-Agents | - Requires deep ML knowledge - Can be heavy for real-time games | Adaptive AI, procedural content, experimental gameplay | Advanced (AI/ML experience) |
| **OpenAI Gym / PettingZoo** | - Great for prototyping RL agents - Good benchmarking environment - Active research ecosystem | - Not production-ready - Integration with game engines takes effort | Research, prototyping, AI training simulations | Advanced (Research/ML focus) |

# Recommendations

**Indie Devs / Beginners** → *A* Pathfinding Project*, *RAIN AI*, or *Unity ML-Agents* (if curious about ML).

**AAA Studios** → *Unreal Engine AI tools* + *Kythera AI* or *Havok AI*.

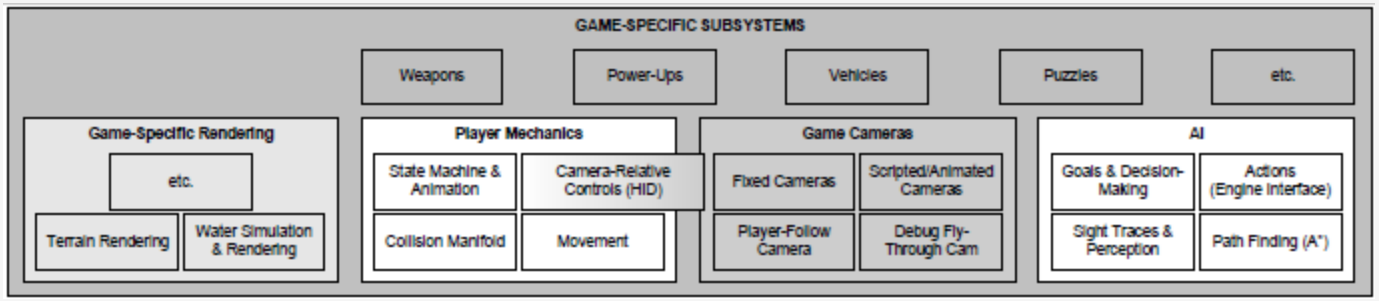
**Researchers / Innovators** → *Unity ML-Agents* + *TensorFlow/PyTorch*.

**Custom Engines** → *Recast & Detour* for navmesh + your own FSM/Behavior Tree system.

# Game-specific subsystems

All of the specific stuff needed for a game

This layer could be considered outside of the game engine itself




On top of the gameplay foundation layer and the other low-level engine components, gameplay programmers and designers cooperate to implement the features of the game itself.

Gameplay systems are usually numerous, highly varied and specific to the game being developed.

Game-specific subsystems include to the mechanics of the player character, various in-game camera systems, artificial intelligence for the control of non-player characters, weapon systems, vehicles, etc.

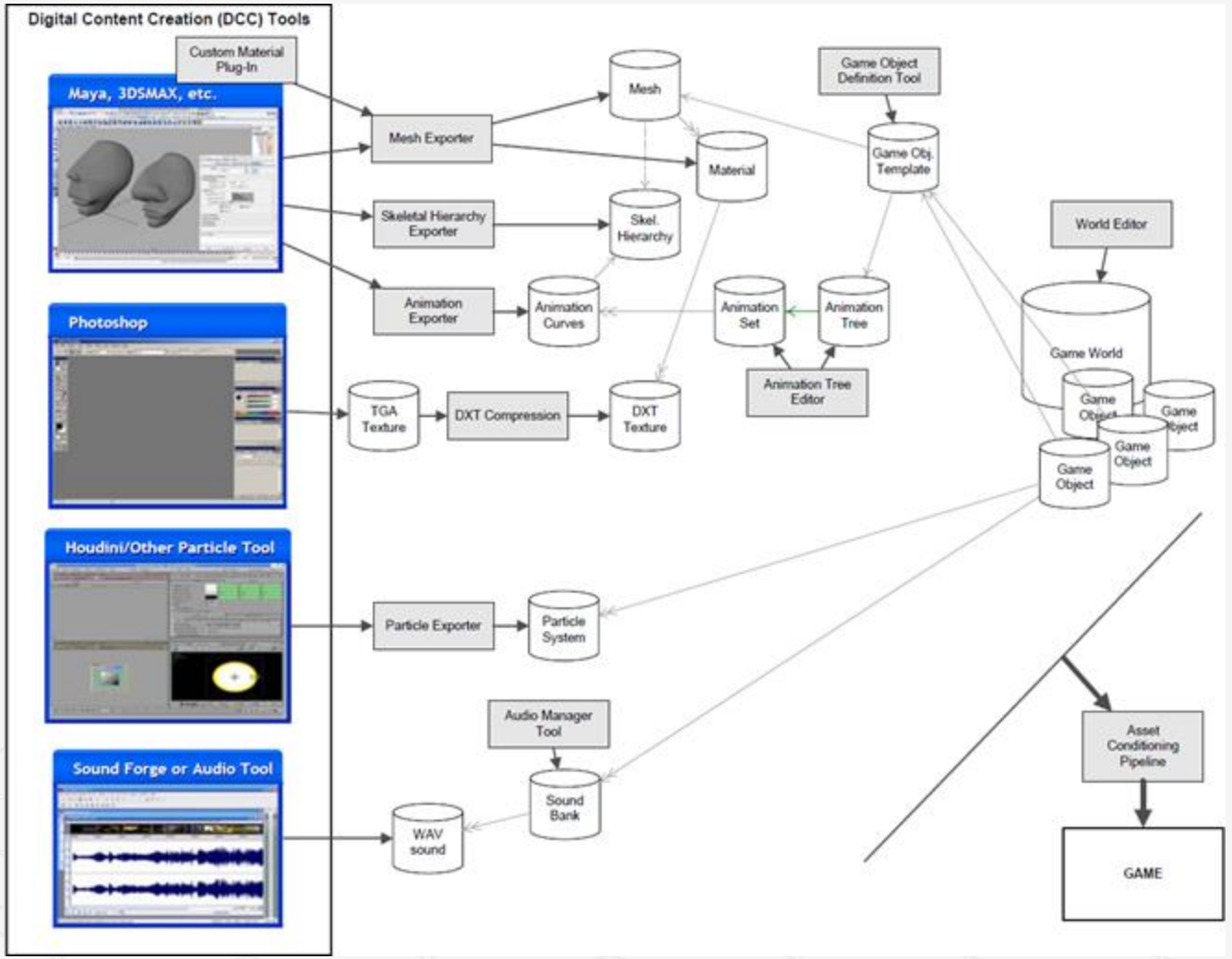
The game-specific subsystems and the gameplay foundations defines the clear line between the game engine and the game.

# Tools and the asset pipeline

Any game engine must be fed a great deal of data, in the form of game assets, configuration files, scripts and so on.

The thicker dark-grey arrows show how data flows from the tools used to create the original source assets all the way through to the game engine itself.

The thinner light-grey arrows show how the various types of assets refer to or use other assets.



# Digital Content Creation

Games are multimedia applications by nature.

Game engines deal with data in many forms.

The tools that the artists use are called *digital content creation* (DCC) applications.

- 3D Meshes, game world layout (Autodesk's Maya and 3ds Max and Pixologic's Zbrush)
- Textures (Adobe's Photoshop)
- Sound (SoundForge)
- Animations

DCC tools need to be easy to use and very reliable

Some types of game data cannot be created using an off-the-shelf DCC app.

# Assets conditioning pipeline

**DCC tools produce a variety of file formats that are not optimized for game**

**Game engines usually store the data in an easy to read and platform specific format**

**Game studio is shipping its game on more than one platform, the intermediate files might be processed differently for each target platform.**

**3D mesh data might be exported to an intermediate format, such as XML, JSON**

**The mesh data might then be organized and packed into a memory image suitable for loading on a specific hardware platform.**

**The ACP does this translation. Every Game Engine has its own Assets Conditioning Pipeline!**

The DCC app's in-memory model of the data is usually much more complex than what the game engine requires.

The DCC application's file format is often too slow to read at runtime, and in some cases it is a closed proprietary format.

# Conditioning assets

## 3D Model/Mesh data

3D Models – Must be properly tessellated

Brush Geometry – A collection of convex hulls with multiple planes (volumetric geometry for older games)

## Skeletal animation data

A *skeletal mesh(skin)* is a special kind of mesh that is bound to a skeletal hierarchy for the purposes of articulated animation.

Must be compressed and converted to properly work

## Audio data

Should convert multiple formats to a single format (like .wav) for the target system

Audio clips are usually exported from Sound Forge or some other audio production tool

## Particle system data

Usually need a custom editing tool within the engine or you can use a third party tool like Houdini.

# Game world editor

The game world is where everything in a game engine comes together.

There are no commercially available game world editors

There are number of commercially available game engines provide good world editors:

- Some variant of the *Radiant* game editor is used by most game engines based on Quake technology.
- The *Half-Life 2* Source engine provides a world editor called *Hammer*.
- *UnrealEd* is the Unreal Engine's world editor. This powerful tool also serves as the asset manager for all data types that the engine can consume.
- *Sandbox* is the world editor in CRYENGINE.

# Resource database

Need a way to store and manage the vast amounts of data

Every asset also carries with it a great deal of *metadata*. The metadata provides the asset conditioning pipeline. For animation:

- A unique id that identifies the animation clip at runtime.
- The name and directory path of the source Maya (.ma or .mb) file.
- The *frame range*—on which frame the animation begins and ends.
- Whether or not the animation is intended to loop.
- The animator's choice of compression technique and level.

Every game engine requires some kind of database to manage all of the metadata associated with the game's assets

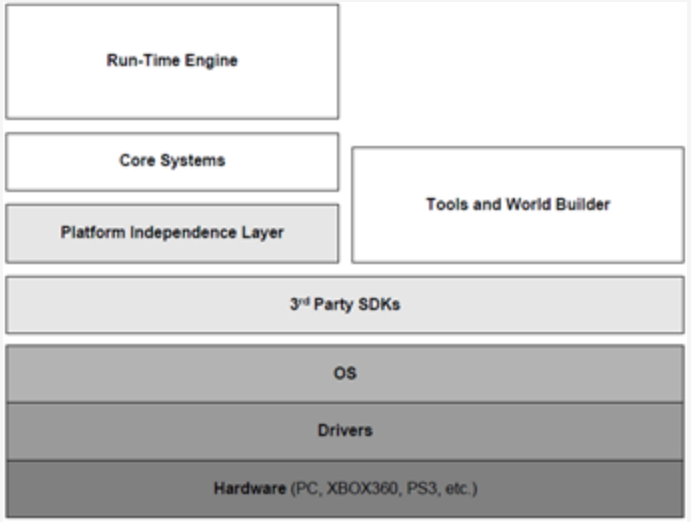
Some companies use relational database MySQL or Oracle

Some companies use text files managed by version control software Subversion, Perforce, or GIT

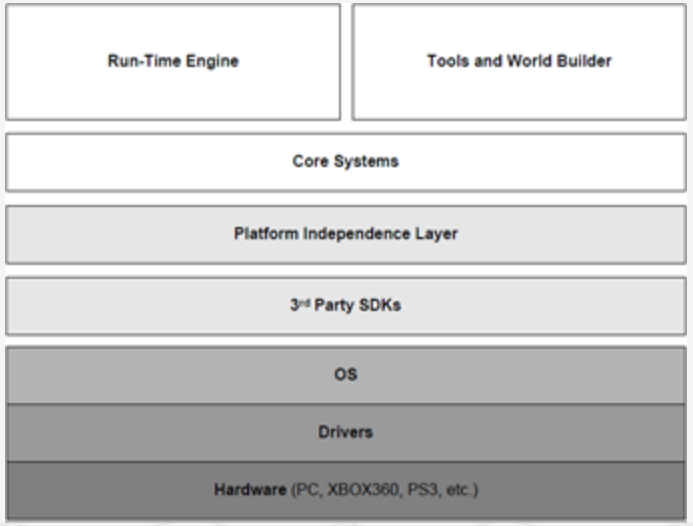
Others also use custom software to edit the metadata. Naughty Dog uses a custom GUI called Builder

# Approaches to Tool Architecture

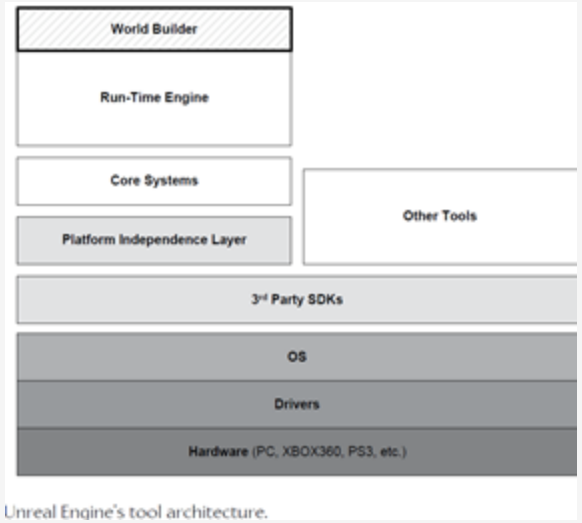
Stand alone




Integrated




Example




Unreal Engine's tool architecture.

# Web-based tools

## Various uses

Asset management
Scheduling
Bug management

## Easier to build

Usually easier to build than a stand alone application
Easier to update without forcing a reinstall
Web apps require no special installation

## If it just needs to present tabular data and have forms – use a web interface

# LAB – Shaders

How to write shaders in Unity

# Writing shaders in Unity

Writing shaders in Unity involves defining how objects are rendered in your scene, including their color, lighting interactions, and visual effects.

Unity offers several approaches to shader creation:

ShaderLab with HLSL/CG

Shader Graph

Surface Shaders (Built-in Render Pipeline)

Render Pipelines:

Unity uses Render Pipelines (Built-in, URP, HDRP) which influence how shaders are written and structured. Ensure your shader targets the correct pipeline.

Shader Semantics:

Special keywords (e.g., : POSITION, : SV_Target) communicate the meaning of variables to the GPU.

# ShaderLab with HLSL/CG

ShaderLab is the traditional and most common method for writing custom shaders in Unity.

Create a Shader File: In your Unity project, right-click in the Project window and select Create > Shader > Standard Surface Shader (or other options depending on your needs, like Unlit Shader for simpler effects).

ShaderLab Structure: The shader file uses ShaderLab, a declarative language that defines the shader's properties, sub-shaders, and passes.

HLSL/CG Programs: Within ShaderLab passes, you write the actual rendering logic using High-Level Shading Language (HLSL) or Cg (a deprecated language, but still supported for compatibility). This is where you define vertex and fragment functions.

Vertex Shader: Transforms mesh data (vertices) into screen-space coordinates.

Fragment Shader: Determines the final color of each pixel based on calculated values, textures, and lighting.

Material Creation: After writing the shader, create a new material (Create > Material) and assign your custom shader to it in the Inspector.

Assign to Object: Apply the material to a 3D object in your scene to see the shader's effect.

# Shader Graph

This visual node-based editor allows you to create shaders without writing code, making it more accessible for beginners.

Create a Shader Graph: Right-click in the Project window and select Create > Shader > [Render Pipeline] > Shader Graph (e.g., URP > Lit Shader Graph).

Node-Based Editing: Drag and drop nodes representing various operations (e.g., color, textures, mathematical functions) and connect them to build your shader's logic.

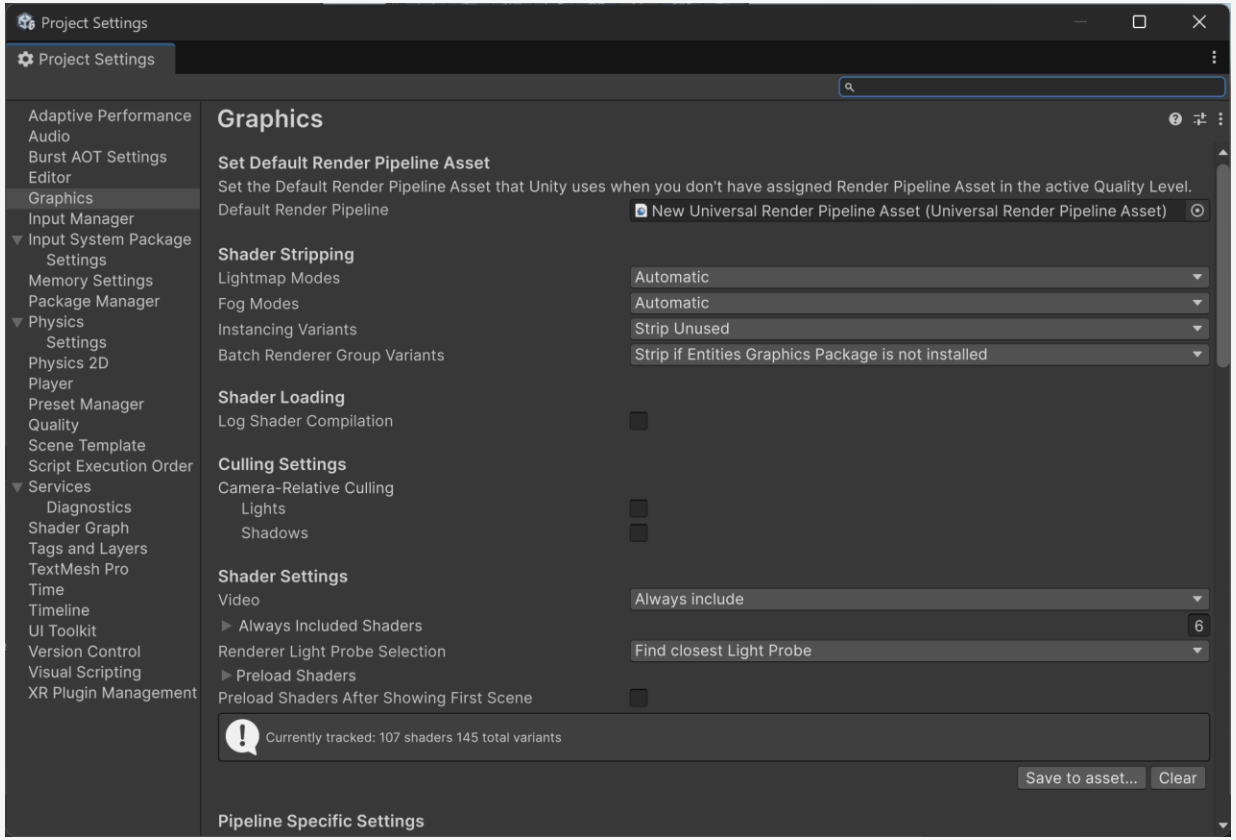
Preview and Apply: The Shader Graph editor provides a real-time preview, and you can easily create materials from your Shader Graph asset and apply them to objects.

# Surface Shaders (Built-in Render Pipeline)

For the Built-in Render Pipeline, Surface Shaders offer a simplified way to write shaders that interact with Unity's lighting system, abstracting away much of the complex lighting calculations.

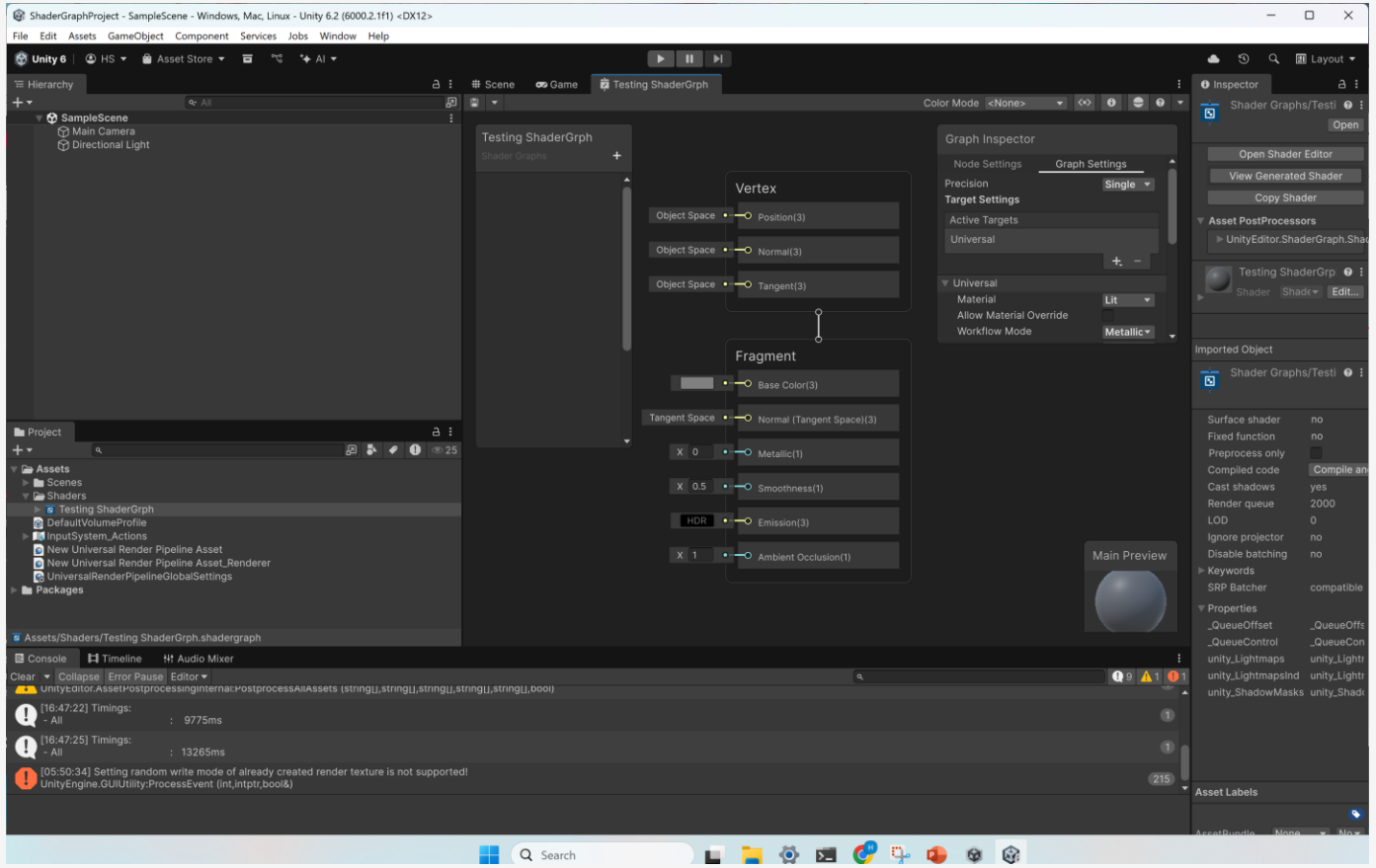
# Hands on Unity Shader Graph

- Create a new 3D Built-in Render Pipeline Project
- BlockNodes, Master Stack
- The block nodes: Vertex blocks (for vertex shader), Fragment blocks (for fragment shaders)
- There are more than 200 nodes
- Install Universal Render Pipeline
- Install Shader Graph from the package manager
- Create a URP asset
  - Right Click on Assets → Create → Renderer → URP Asset
- Drag the UniversalRenderPipelineAsset to “Scriptable Render Pipline Settings” of “Graphics” in the project settings.



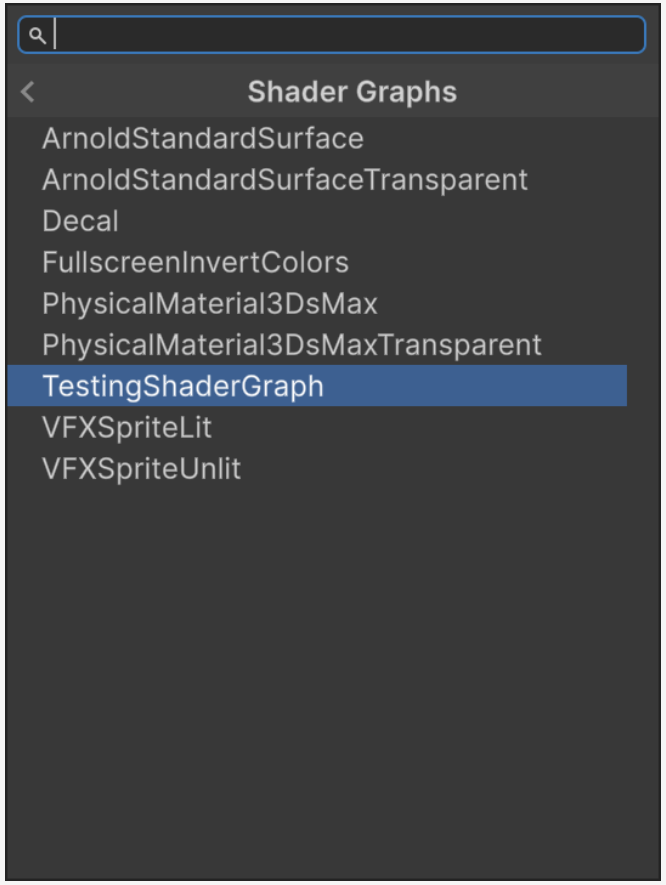
# Add a Lit Shader Graph

- Create a folder called “Shaders” under “Assets”
- Right Click on the “Shaders” folder → Create → ShaderGraph → URP →Lit Shader Graph
- Rename it to TestingShaderGraph, double click and open it
- If you right click on “Main Preview”, you can change the sphere to different geometries.



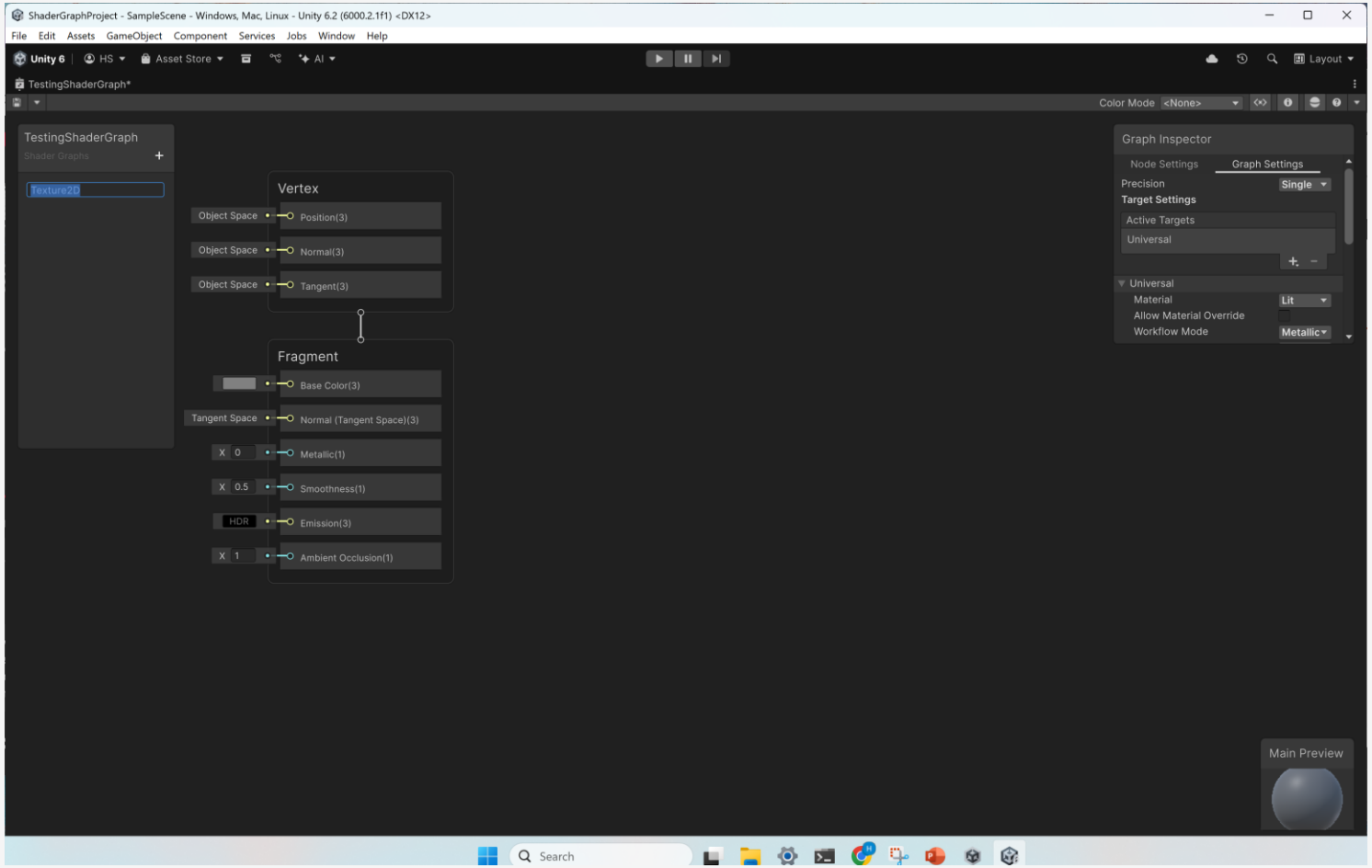
# Add a material

- Create a new Material under “Shaders”, call it “TestingMaterial”
- Change its shader to our “TestingShaderGraph”
- In the “Hierarchy” window, create a new 3D Gameobject “Cube” and apply the material.
- Go back to TestingShaderGraph
- Press Shift + Space to maximize the window

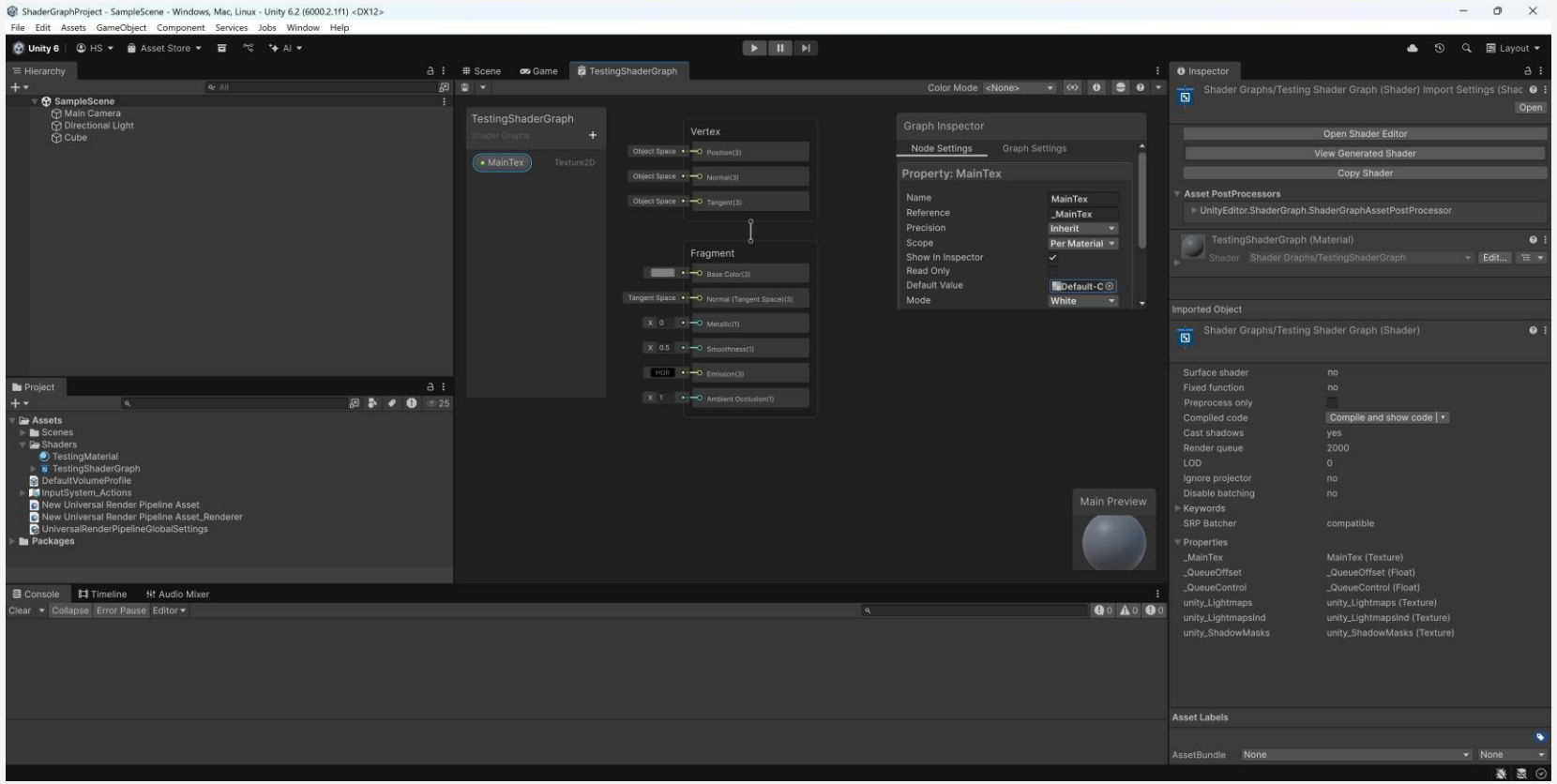


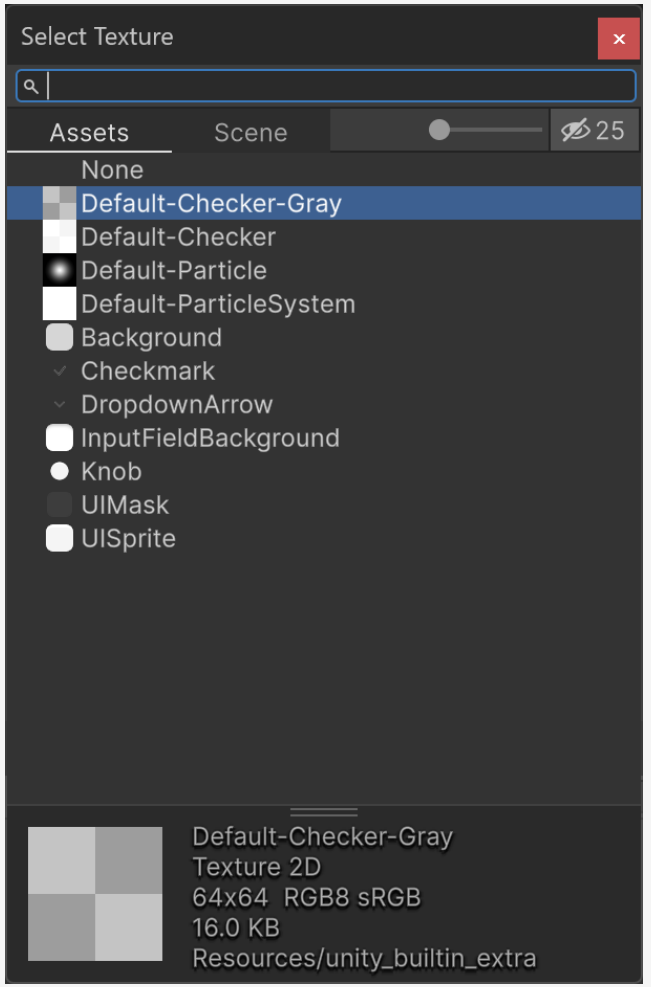
# Properties

- Add a Texture2D property
- Change its name to MainTex
- Notice that its reference is changed to _MainTex (in case you want to use it in the code)
- Save your asset!



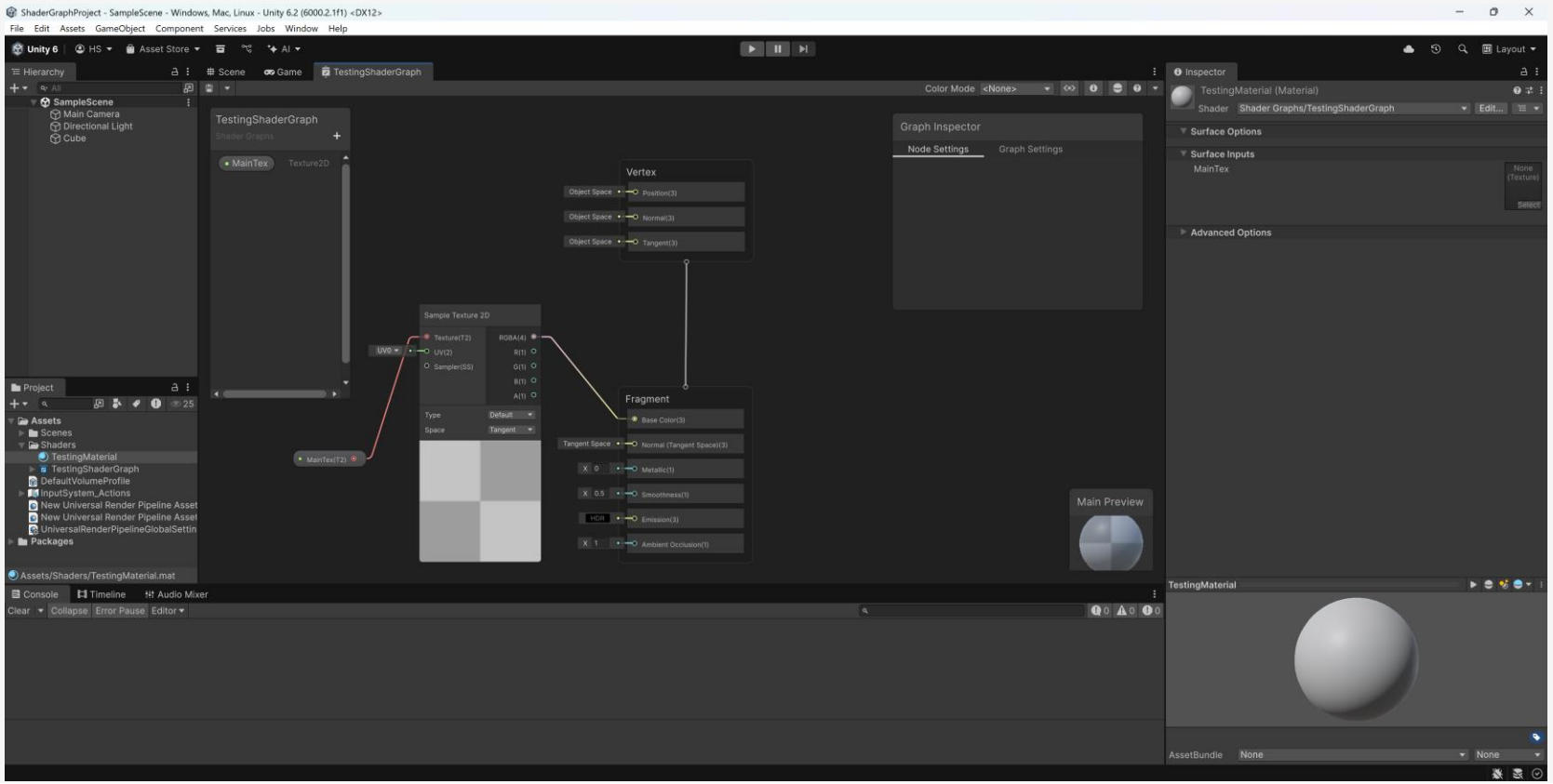
# Add a texture to the “MainTex” property (Default Value)



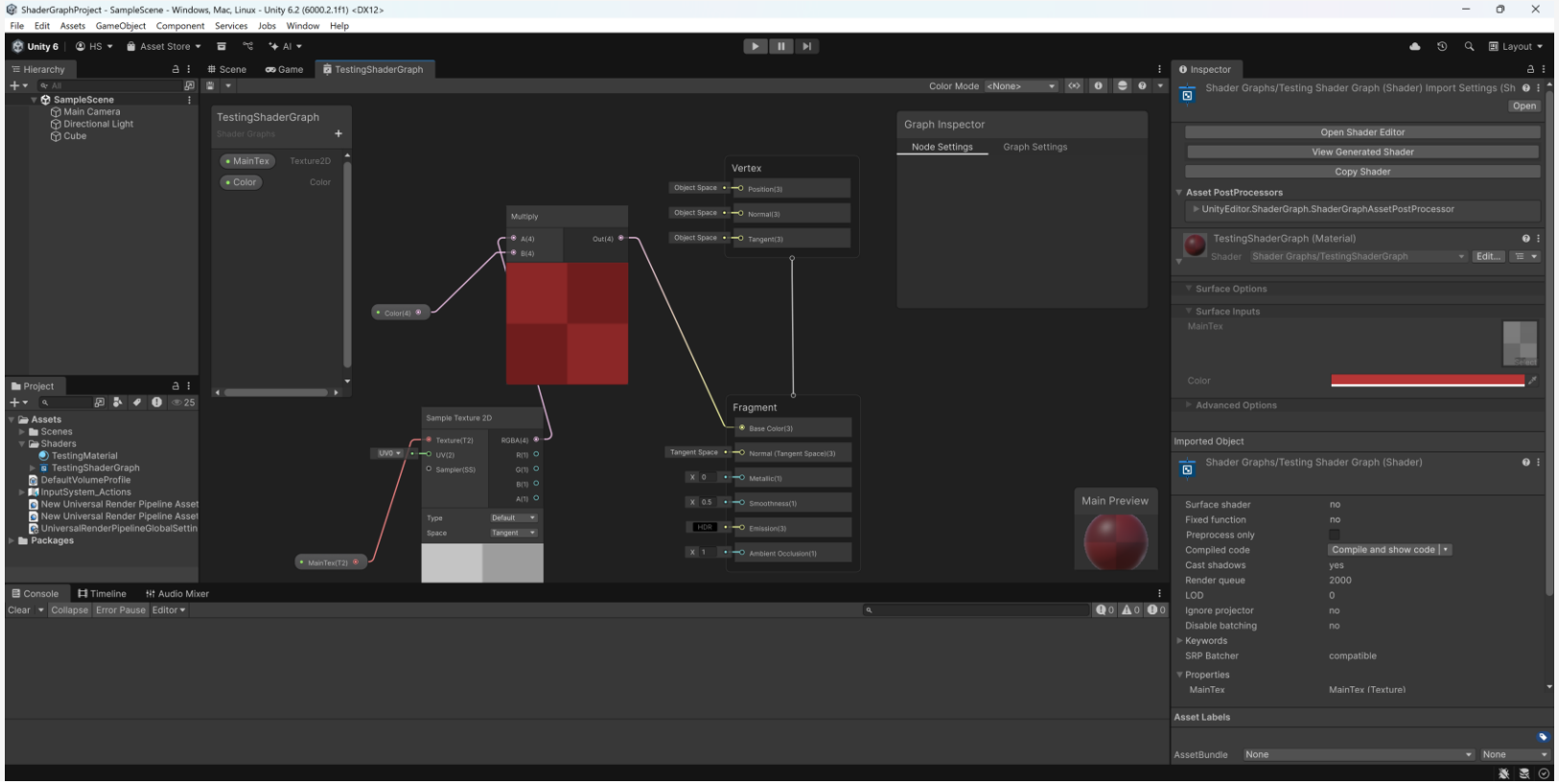
# Add a “Sample Texture 2D” Node

- Drag “MainTex” property to the graph
- Create a “Sample Texture 2D” node under  Input (press space to add a node)
- Connect the RGBA node to the “Base Color”

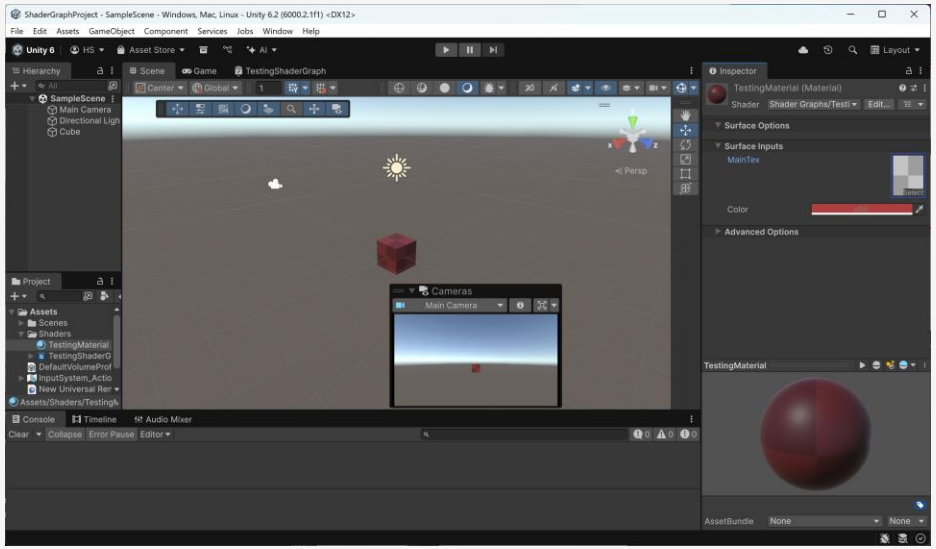


# Tint our object with a new color property

- Add a new “Color” property
- Drag it onto the board
- Add a “Multiply” node, to multiply color to our texture
- Change the “Default” color to red(or any color)
- Connect the nodes



# Update the “MainTex” and “Color” in the “TestingMaterial”



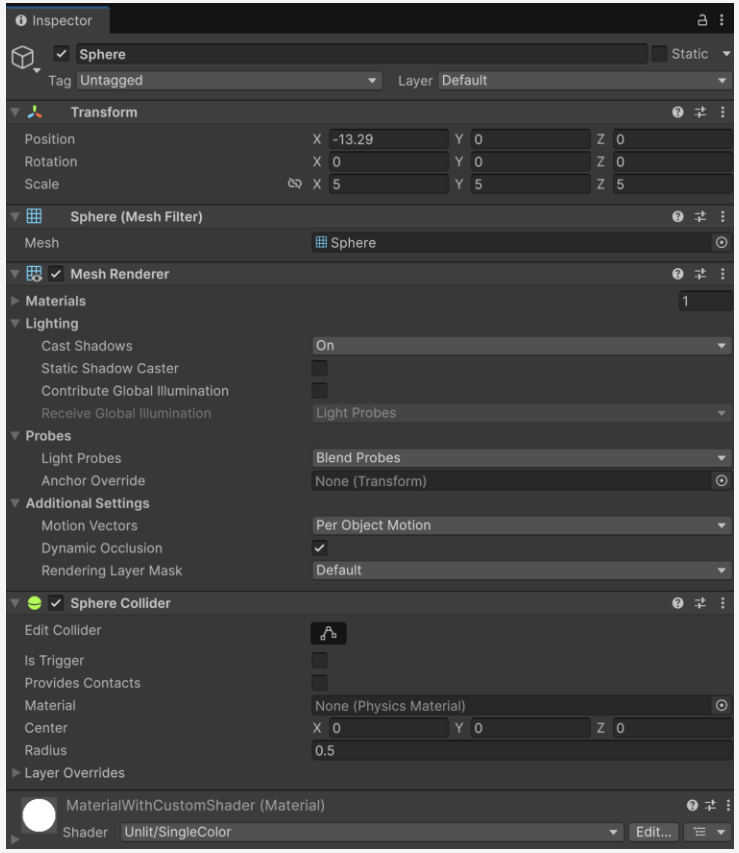
# Single color shader example in the Built-In Render Pipeline

```
Shader "Unlit/SingleColor"
{
    Properties
    {
        // Color property for material inspector, default to white
        _Color("Main Color", Color) = (1,1,1,1)
    }
        SubShader
    {
        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            // vertex shader
            // this time instead of using "appdata" struct, just spell inputs manually,
            // and instead of returning v2f struct, also just return a single output
            // float4 clip position
            float4 vert(float4 vertex : POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, vertex);
            }

        // color from the material
        fixed4 _Color;

        // pixel shader, no inputs needed
        fixed4 frag() : SV_Target
        {
            return _Color; // just return it
        }
        ENDHLSL
    }
    }
}
```



# Built-in shader variables reference

https://docs.unity3d.com/6000.1/Documentation/Manual/SL-UnityShaderVariables.html

| Name | Value |
|---|---|
| UNITY_MATRIX_MVP | Current model * view * projection matrix. |
| UNITY_MATRIX_MV | Current model * view matrix. |
| UNITY_MATRIX_V | Current view matrix. |
| UNITY_MATRIX_P | Current projection matrix. |
| UNITY_MATRIX_VP | Current view * projection matrix. |
| UNITY_MATRIX_T_MV | Transpose of model * view matrix. |
| UNITY_MATRIX_IT_MV | Inverse transpose of model * view matrix. |
| unity_ObjectToWorld | Current model matrix. |
| unity_WorldToObject | Inverse of current world matrix. |

| Name | Type | Value |
|---|---|---|
| _WorldSpaceCameraPos | float3 | World space position of the camera. |
| _ProjectionParams | float4 | x is 1.0 (or –1.0 if currently rendering with a flipped projection matrix), y is the camera's near plane, z is the camera's far plane and w is 1/FarPlane. |
| _ScreenParams | float4 | x is the width of the camera's target texture in **pixels** , y is the height of the camera's target texture in pixels, z is 1.0 + 1.0/width and w is 1.0 + 1.0/height. |
| _ZBufferParams | float4 | •Used to linearize Z buffer values.x: 1 - far/near, or –1 + far/near if UNITY_REVERSED_Z is set to 1. For more information about UNITY_REVERSED_Z, refer to Branch based on platform features.<br>•y: far/near, or 1 if UNITY_REVERSED_Z is set to 1. For more information about UNITY_REVERSED_Z, refer to Branch based on platform features.<br>•z: x / far<br>•w: y / far |
| unity_OrthoParams | float4 | x is orthographic camera's width, y is orthographic camera's height, z is unused and w is 1.0 when camera is orthographic, 0.0 when perspective. |
| unity_CameraProjection | float4x4 | Camera's projection matrix. |
| unity_CameraInvProjection | float4x4 | Inverse of camera's projection matrix. |
| unity_CameraWorldClipPlanes[6] | float4 | Camera frustum plane world space equations, in this order: left, right, bottom, top, near, far. |