CMSC 425: Lecture 2
Computer Game and Graphics System Architectures

Reading: The first half of the lecture is taken from Chapt 1 of Gregory, *Game Engine Architecture*. The second half comes from standard computer graphics texts.

Origins of Computer Game Engines: A large computer game is a significant technical undertaking, involving a large number of interacting components. Of course, not all computer games require the same level of complexity. Different genres of games require different capabilities. The combination of components used for a simple casual 2-dimensional game is very different from a high-end AAA 3-dimensional game.

One way to better understand the software structure underlying a generic game is to understand the structure of a typical game engines. Game engines arose in the mid-1990s. In particular, the software for the popular game *Doom* provided a separation between:

- core game components (such as the rendering system, collision detection system, audio system)
- art assets (models, textures, animations)
- rules of play

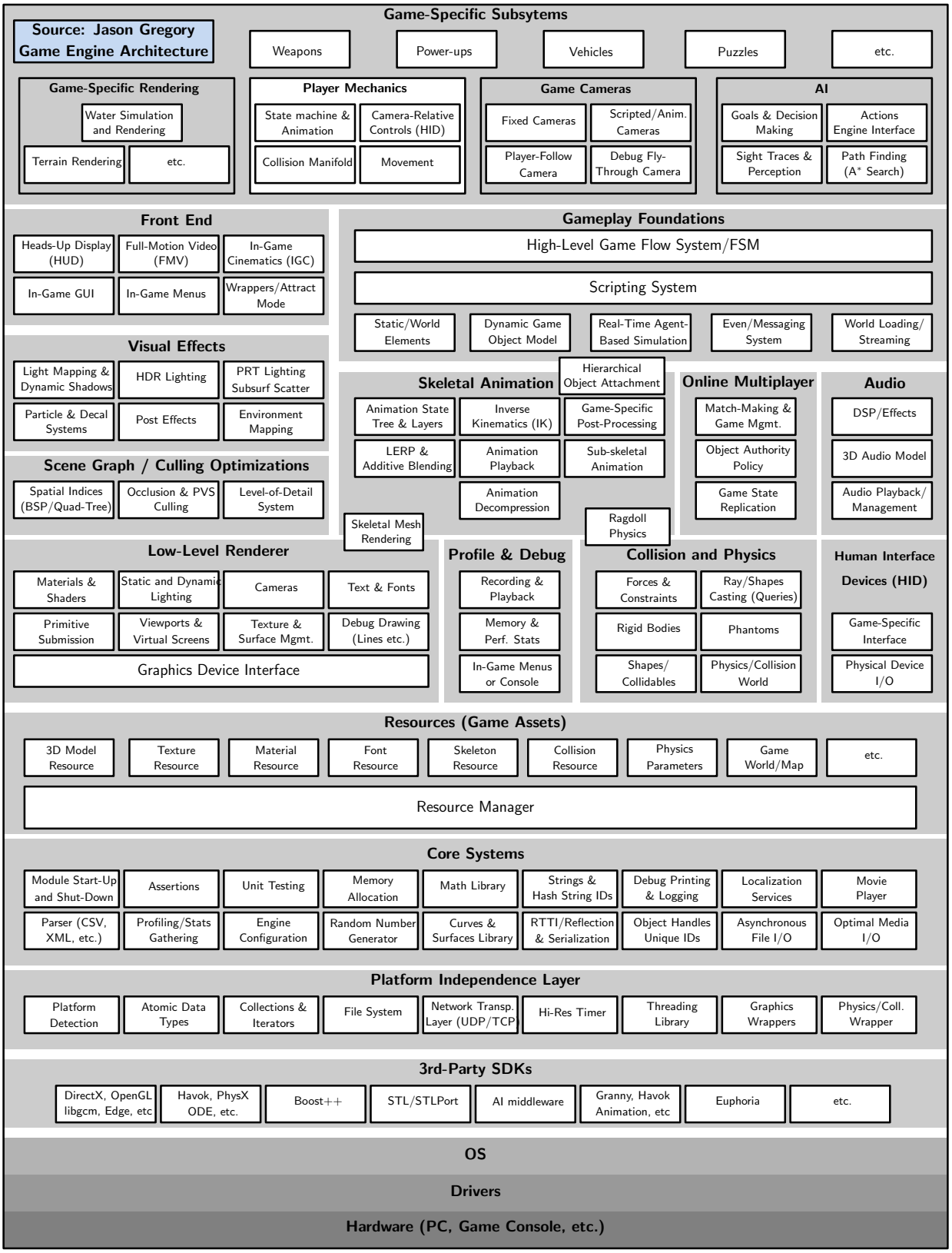
This separation made it easy for users to modify, or "modding," the game, and provided a framework for adding new elements. This model was extended to other games, including *Quake*, *Unreal*, and *Unreal Tournament* (all FPS games). At some point, these simple "modding systems" became generic enough that it was possible to implement a wide variety of very different games based on a common core set of components, the *game engine*. Examples of modern game engines include *Unity 3D* and *Unreal Engine 4*.

Game engines vary along a spectrum of ease of use and flexibility. Simple game engines can generate only a single type of game (e.g., 2-dimensional games), but are generally easy to pick up and use. Complex game engines can generate a great variety of games, but it can take quite a bit of time to master their various capabilities.

The following is a summary of the basic components of a modern game engine. We think of the engine as being designed in a number of layers, ranging from the lower levels (hardware and operating system) up to the higher levels (game specific entities like rules). Here is a summary of the levels, from low to high. These are illustrated in the figure below.

System: This includes low-level software for interacting with the operating system on which the game engine runs as well as the target system on which the game executes. Target systems can include general personal computers (running, say, Microsoft Windows, Linux, or Mac OS), game consoles (e.g., XBox, Playstation, Wii), or mobile devices (e.g., hand-held game consoles, tablets, and smart phones).

Third-Party SDKs and Middleware: These are libraries and sofware development toolkits (SDKs), usually provided from a third party. Examples include graphics (e.g., OpenGL and DirectX), physics (e.g., Havok, PhysX, and Bullet), basic algorithms and data structures (e.g., Java Class Library, C++ STL, Boost++), character animation (e.g., Granny), networking support (e.g., Unix sockets).


Game-Specific Subsytems
Source: Jason Gregory Game Engine Architecture
Weapons
Power-ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
Water Simulation and Rendering
Terrain Rendering
etc.
Player Mechanics
State machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Anim. Cameras
Player-Follow Camera
Debug Fly-Through Camera
AI
Goals & Decision Making
Actions Engine Interface
Sight Traces & Perception
Path Finding (A* Search)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers/Attract Mode
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static/World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Even/Messaging System
World Loading/ Streaming
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting Subsurf Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP & Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback/ Management
Scene Graph / Culling Optimizations
Spatial Indices (BSP/Quad-Tree)
Occlusion & PVS Culling
Level-of-Detail System
Skeletal Mesh Rendering
Ragdoll Physics
Low-Level Renderer
Materials & Shaders
Static and Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture & Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profile & Debug
Recording & Playback
Memory & Perf. Stats
In-Game Menus or Console
Collision and Physics
Forces & Constraints
Ray/Shapes Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings & Hash String IDs
Debug Printing & Logging
Localization Services
Movie Player
Parser (CSV, XML, etc.)
Profiling/Stats Gathering
Engine Configuration
Random Number Generator
Curves & Surfaces Library
RTTI/Reflection & Serialization
Object Handles Unique IDs
Asynchronous File I/O
Optimal Media I/O
Platform Independence Layer
Platform Detection
Atomic Data Types
Collections & Iterators
File System
Network Transp. Layer (UDP/TCP)
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd-Party SDKs
DirectX, OpenGL libgcm, Edge, etc
Havok, PhysX ODE, etc.
Boost++
STL/STLPort
AI middleware
Granny, Havok Animation, etc
Euphoria
etc.
OS
Drivers
Hardware (PC, Game Console, etc.)

Platform Independence Layer: Since most games are developed to run on many different platforms, this layer provides software to translate between game specific operations and their system-dependent implementations.

Core System: These include basic tools necessary in any software development environment, including assertion testing, unit testing, memory allocation/deallocation, mathematics library, debugging aids, parsers and serializers (e.g., for xml-based import and export), file I/O, video playback.

Resource Manager: Large graphics programs involve accessing various resources, such as geometric models for characters and buildings, texture images for coloring these geometric models, maps representing the game's world. The job of the resource manager is to allow the program to load these resources. Since resources may be compressed to save space, this may also involve decompression.

Rendering Engine: This is one of the largest and most complex components of any real-time 3-dimensional game. This involves all aspects of drawing, and may involve close interaction with the graphics hardware, or *graphics processing unit* (GPU), for the sake of enhanced efficiency.

Low-Level Renderer: This comprises the most basic elements of producing images. Your program interacts with the GPU by asking it to render *objects*. Each object may be as simple as a single triangle but is more typically a mesh consisting of many triangular/polygonal elements. Objects are specified according to their coordinates in 3-dimensional space. Your program also informs the GPU what colors (or what image textures) to apply to these objects, where lights are positioned, and where the camera is positioned.

It is then the job of the GPU to perform the actual rendering (projection, coloring, shading) of the objects. In particular, it determines where each object projects onto the 2-dimensional image plane, which objects are visible and which are hidden from view, what is the color and brightness of each object. Your program needs to convey all this information to the GPU. This also includes elements like displaying text messages and subdividing the window into subwindows (called *viewports*) for the purposes of showing status information or maps. (Further details are provided later in this lecture.)

Graphics Device Interface: Since the graphics device requires updating 30–100 times per second, but some operations (such as displaying messages to the user) occurs at a significantly different time scale (of multiple seconds), these components shield the programmer from some of the low-level timing issues when dealing with the graphics system.

Scene Graph: Game entities are naturally organized into hierarchical structures. This is true for dynamic and static objects. For example, a human body consists of a head, torso, arms, legs; an arm consists of a hand, lower-arm, and upper-arm; a hand consists of fingers. Thus, there is a natural structure in the form of a rooted tree.

In general, all the entities of the games world can be represented in a large tree, where the root represents the entire world, and the nodes of the tree implicitly represent the subtrees that are descended from these nodes. This makes it possible

to perform operations easily on an entire portion of the tree. For example, we could "render the objects rooted at node u" or "rotate the object rooted at node v." We can also create multiple instances, as in "create 200 instances of the zombie object at node z."

This software component is responsible for creating, modifying, rendering, manipulating, and deleting elements of the scene graph. Another feature of using a scene graph is that it allows us to remove, or *cull*, entities that are not visible to the camera. For example, if the camera is located in a room represented by some node v, we need only render the objects lying within this room or that are visible through the room's doors and windows. Because game worlds are so large and complex, efficient rendering demands that we only attempt to draw the things that might be visible to the camera.

Visual Effects: This includes support for a number of complex effects such as:

- particle systems (which are used for rendering smoke, water, fire, explosions)
- decal systems (for painting bullet holes, damage scratches, powder marks from explosions, foot prints, etc)
- complex lighting, shadowing, and reflections

Others: This includes visual elements of the user interface, such as displaying menus or debugging aids (for the developer) and video playback for generating the back story.

Collisions and Physics: These components simulate the movement of objects over time, detect when objects collide with one another, and determine the appropriate response in the event of a collision (like knocking down the houses where the little pigs live). Except in very simple physical phenomena, like a free-falling body, physical simulation can be very difficult. For this reason, it is often handled by a third-party physics SDK.

Animation: While the game may direct a character to move from one location to another, the job of the animation system is to make this motion look natural, for example, by moving the arms and legs in a manner that is consistent with normal walking behavior. The typical process for most animals (including humans) involves developing a skeletal representation of the object and wrapping flesh (which is actually a mixture of skin and clothing) around this skeleton. The skeleton is moved by specifying the changes in the angles of the various joints. This approach is called a *skin and bones* representation. Another issue is blending between animations, such as smoothly transitioning from standing to walking to running animations.

Input Handlers: These components process inputs from the user, including keyboard, mouse, and game controller inputs. Some devices also provide feedback to users (such as the vibration in some game controllers). Modern vision based systems, such as the XBox Kinect, add a whole new level of complexity to this process.

Audio: Audio components handle simple things like playback for background music, explosions, car engines, tire squealing, but they may also generate special audio effects, such as stereo effects to give a sense of location, echo effects to give a sense of context (inside versus outside), and other audio effects (such as creating a feeling of distance by filtering out high-frequency components).

Multiplayer/Networking: Multiplayer and online games require a number of additional supporting components. For example, multiplayer games may have a split-screen capability. Online games require support for basic network communication (serialization of structures into packets, network protocol issues, hiding network latency, and so on). This also includes issues in the design of game servers, such as services for maintaining registered users and their passwords, matching players with one another, and so on.

Gameplay Foundation System: The term *gameplay* refers to the rules that govern game play, that is, the player's possible actions and the consequences of these actions. Since this varies considerably from one game to another, designing a system that works for a wide variety of games can be quite daunting. Often, these systems may be designed to support just a single genre of games, such as FPS games. There are a number of subcomponents:

Game Worlds and Object Models: This constitutes the basic entities that make up the game's world. Here are some examples:

- static background objects (buildings, terrain, roads)
- (potentially) dynamic background objects (rocks, chairs, doors and windows)
- player characters (PCs) and non-player characters (NPCs)
- weapons and projectiles
- vehicles
- graphics elements (camera and lights)

The diversity of possible game objects is a major challenge to programmers trained in object-oriented methods. Objects share certain universal qualities (e.g., they can be created and destroyed), common qualities (e.g., they can be rendered), and more specialized qualities (e.g., gun-like objects can be shot, chair-like objects can be sat on).

Event System: Game objects need to communicate with one another. This is often handled by a form of message passing. When a message arrives, we can think of it as implicitly signaling an event to occur, which is to be processed by the entity. Game objects can register their interest in various events, which may affect their behavior. (For example, characters need to be made aware of explosions occurring in their vicinity. When such an explosion occurs, the system informs nearby characters by sending them a message..."you're toast, buddy.")

Scripting System: In order to allow game developers to rapidly write and test game code, rather than have them write in a low-level programming language, such as C++, it is common to have them produce their code in a scripting language, like Python. A sophisticated scripting system can handle the editing of scripts and reloading a script into a game while it is running.

Artificial Intelligence: These components are used to control the behavior of non-player characters. They are typically modeled as AI *agents*. As we will discuss later, an agent is an object that can sense its environment, maintain a memory (or state), and can respond in potentially complex ways depending on the current input and state. One of the elements that we can lump under the AI topic is navigation (although it may not involve very much "intelligence"). This deals with computing

shortest paths for both player and non-player characters to move from one location to another while avoiding obstacles or achieving other objectives (e.g., staying out of sight of the enemy in a military game).

Game Specific Systems: This is a catch-all for any components that are so specific to a given game that they don't fall into one of the other categories. This may include aspects like the mechanics controlling a player's state, the algorithms by which a camera moves throughout the world, the specific goals of the game's non-player characters, the properties of various weapons, and so on.

Interactive 3-dimensional Graphics: (Optional material)

In order to get a quick jump into game development, we will start by discussing the basic elements of interactive computer graphics systems. This will require that we understand a bit about how graphics processing units work and will lead us eventually into a discussion of OpenGL.

Anyone who has played a computer game is accustomed to interaction with a graphics system in which the principal mode of rendering involves 3-dimensional scenes. Producing highly realistic, complex scenes at interactive frame rates (at least 30 frames per second, say) is made possible with the aid of a hardware device called a *graphics processing unit*, or *GPU* for short. GPUs are very complex things, and we will only be able to provide a general outline of how they work.

Like the CPU (central processing unit), the GPU is a critical part of modern computer systems. (See Fig. 1 for a schematic representation.) Because of its central importance, the GPU is connected to the CPU and the system memory through a high-speed data transfer interface, which is current systems architecture terminology, is called the *north bridge*. The GPU has its own memory, separate from the CPU's memory, in which it stores the various graphics objects (e.g., vertex coordinates and textures) that it needs in order to do its job. The GPU is highly parallel, and it has a very high capacity connection with its memory. Part of this memory is called the *frame buffer*, which is a dedicated chunk of memory where the pixels associated with your monitor are stored.

Traditionally, GPUs are designed to perform a relatively limited fixed set of operations, but with blazing speed and a high degree of parallelism. Modern GPUs are *programmable*, in that they provide the user the ability to program various elements of the graphics process. For example, modern GPUs support programs called *vertex shaders* and *fragment shaders*, which provide the user with the ability to fine-tune the colors assigned to vertices and fragments.

Recently there has been a trend towards what are called *general purpose GPUs* (GPGPUs), which can perform not just graphics rendering, but general scientific calculations on the GPU. Since we are interested in graphics here, we will focus on the GPUs traditional role in the rendering process.

The Graphics Pipeline: The key concept behind all GPUs is the notion of the *graphics pipeline*. This is conceptual tool, where your user program sits at one end sending graphics commands to the GPU, and the frame buffer sits at the other end. A typical command from your program might be "draw a triangle in 3-dimensional space at these coordinates." The job of the graphics system is to convert this simple request to that of coloring a set of pixels on your

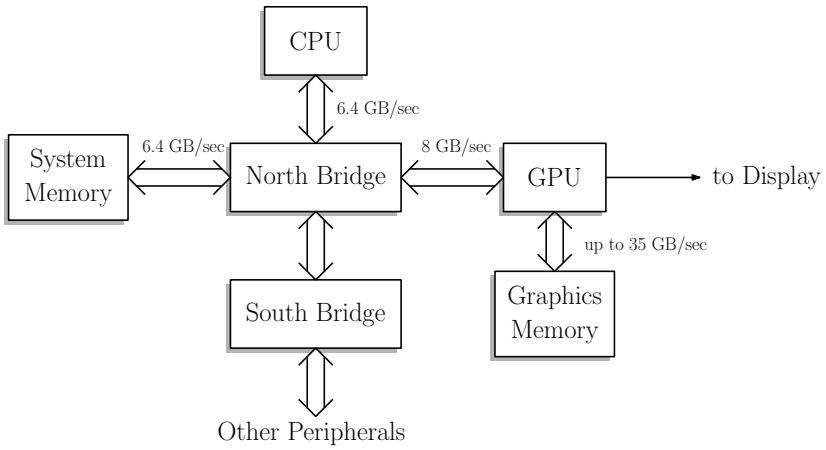


Fig. 1: Architecture of a simple GPU-based graphics system. (Adapted from NVIDIA GeForce documentation.)

display. The process of doing this is quite complex, and involves a number of stages. Each of these stages is performed by some part of the pipeline, and the results are then fed to the next stage of the pipeline, until the final image is produced at the end.

Broadly speaking the pipeline can be viewed as involving a number of stages (see Fig. 2). Geometric objects, called *primitives*, are introduced to the pipeline from your program. Objects are described in terms of vectors in 3-dimensional space (for example, a triangle might be represented by three such vectors, one per vertex).

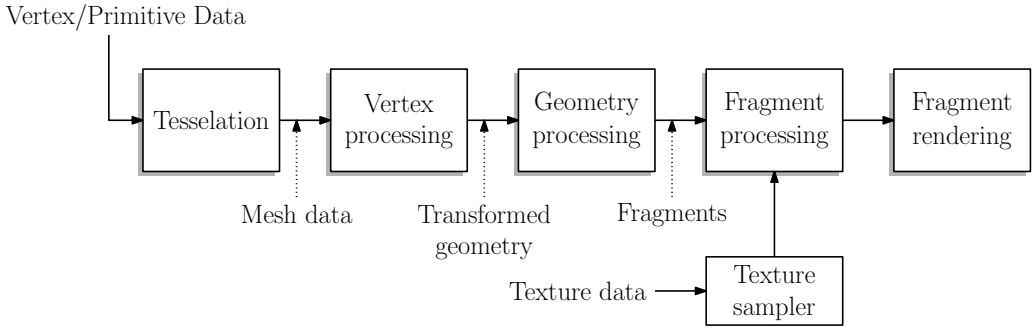


Fig. 2: Stages of the graphics pipeline.

Tesselation: Converts higher-order primitives (such as surfaces), displacement maps, and mesh patches to 3-dimensional vertex locations and stores those locations in *vertex buffers*, that is, arrays of vertex data.

Vertex processing: Vertex data is *transformed* from the user's coordinate system into a coordinate system that is more convenient to the graphics system. For the purposes of this high-level overview, you might imagine that the transformation projects the vertices of the three-dimensional triangle onto the 2-dimensional coordinate system of your screen, called *screen space*.

Geometry processing: This involves a number of tasks:

- *Clipping* is performed to snip off any parts of your geometry that lie outside the viewing area of the window on your display.
- *Back-face culling* removes faces of your mesh that lie on the side of an object that faces away from the camera.
- *Lighting* determines the colors and intensities of the vertices of your objects. Lighting is performed by a program called a *vertex shader*, which you provide to the GPU.
- *Rasterization* converts the geometric shapes (e.g., triangles) into a collection of pixels on the screen, called *fragments*.

Texture sampling: Texture images are sampled and smoothed and the resulting colors are assigned to individual fragments.

Fragment Processing: Each fragment is then run through various computations. First, it must be determined whether this fragment is *visible*, or whether it is hidden behind some other fragment. If it is visible, it will then be subjected to coloring. This may involve applying various coloring textures to the fragment and/or color blending from the vertices, in order to produce the effect of smooth shading.

Fragment Rendering: Generally, there may be a number of fragments that affect the color of a given pixel. (This typically results from translucence or other special effects like motion blur.) The colors of these fragments are then blended together to produce the final pixel color. Fog effects may also be involved to alter the color of the fragment. The final output of this stage is the *frame-buffer image*.