PA199
Advanced Game Design

Lecture 5

Game Engine Architectures

Dr. Fotis Liarokapis

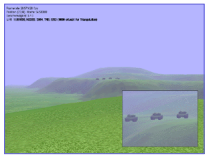
23rd March 2016

Game Engine Programming

- Game engine programming supports the execution of the game
- Game Engine development is one of the basic jobs within the computer games industry

What is a Game Engine?

- A game engine is an open, extendable software system on which a computer game can be built
- A game engine is free from any function, parameter, variable, class or data structure that could be considered part of an actual game [Zerbst et al. 2003]



Game Engines

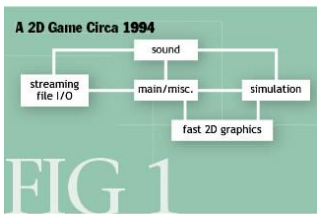
- Developed to abstract away underlying aspects of a game
- Enables reuse of code
- Facilitates porting code to other hardware platforms

Game Engine Purpose

- Provision of a generic infrastructure for game creation
  - i.e. I/O and resource/asset management facilities
- A game engine does not provide data or functions that could be associated with any game or other application of the game engine
- Provision of a glue layer that connects the engine's component parts, which sets a game engine apart from an API (a set of reusable components that can be transferred between different games), making it more than the sum of its components and sub-systems

Engine Development History

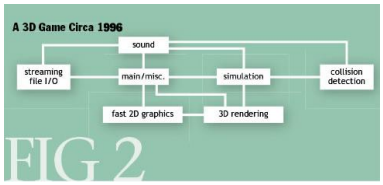
- Game Engines started simple:
  - Sound
  - File I/O
  - Simulation
  - Graphics
  - Misc



[Blow2004]

Engine Development History .

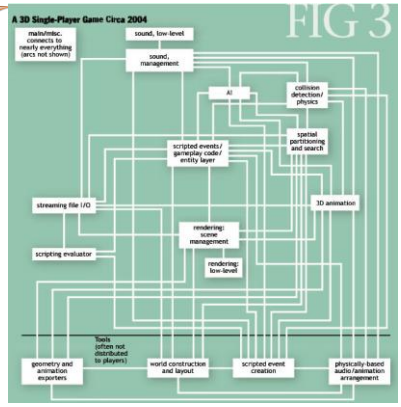
- Game Engines complexity quickly increased:
  - Sound
  - File I/O
  - Simulation
  - Graphics
  - Rendering
  - Collision detection
  - Misc



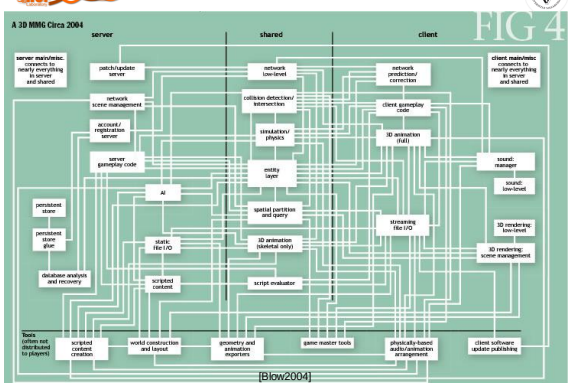
[Blow2004]

Engine Development History ..

- Modern engines are complex modular systems
- Engines now also include content creation tools



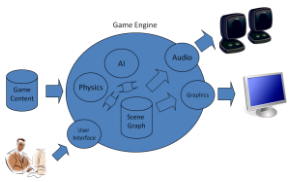
[Blow2004]



[Blow2004]

Game Engine Modules

- Contains a number of software modules integrated together to provide the gaming experience:
  - Graphics
  - Physics
  - Sound
  - Scripting
  - Animation
  - Artificial Intelligence
  - Networking
  - User Interface



Graphics Module

- Provides:
  - Abstraction of the GPU in the hardware
  - Ability to render the geometric content created by artists at the highest level of fidelity possible
- Incorporates advanced real-time mathematical models of how light interacts with the surface of an object
- Modern GPUs are programmable allowing for very sophisticated effects

Graphics Module .

- Higher level interface, tuned to a particular graphics and game type
  - Sprite-based, isometric, full 3D, etc
- Can deal with higher level modelling concepts
  - Sprites, solids, characters (articulated), etc
- Handles more complicated display aspects
  - Mini map, multiple views, overlays, special effects, etc
- Some of these engines are for sale or available on the web
- Often remade or heavily tuned for each game
  - Too much time and money is spent on this

Physics Module

- Implements Newtonian model of physics for the game environment objects
- Gives the game object's mass, force and velocity and uses collision detection methods to provide animation
- Often uses simplifications of physics models to simulate the physics in order to provide real-time responses

Physics Module .

- Limited or non-existent in simple games
- Some commercial/open source engines:
  - ODE, Havok, Tokamak, etc
- Physics *hardware*
  - NVidia/Ageia PhysX
- Physics is more and more integrated into the gameplay and game subsystems
  - Physics-based animation
  - Interaction with objects using physics

Sound Module

- Manages the components of the soundtrack content developed by musicians and sound designers
- Executes sounds based on triggers in the environment from explicit execution, time events or interactions between game objects
  - i.e. collisions

Sound Module .

- Function of sound
  - Effects to enhance reality
  - Ambience
  - Clues about what to do
  - Clues about what is about to happen (but be careful)
- Sound formats
  - Wave (high quality, lots of memory, fast)
  - MP3 (high quality, compressed, slower)
  - Midi (lower quality, very low storage, limited, adaptable)
  - CD (Very high quality, fast, limited to background music)

Sound Module ..

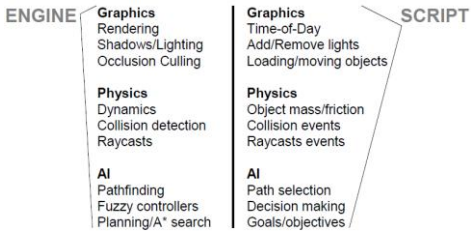
- Simultaneous sounds
  - Mixers (hidden in the HAL)
  - Buffer management
  - Streaming sound
- Special features
  - Positional 3D sound (possibly with Dolby surround)
- Important for clues
  - Adaptive music (DirectMusic)

Scripting Module

- A language add-on to the engine that exposes internal engine architecture
- Allows designers to glue the game together by implementing the gameplay mechanics
  - i.e. events, AI, rewards, etc.

Scripting Module .

- What belongs in a script and what belongs in the engine?



Scripting Module ..

- Advantages:
  - Easy control of many (or all) features in the game engine
  - Scripting language often provides full OO control
    - i.e. Lua
  - Promotes data-driven design
- Disadvantages:
  - Performance
  - Development support tools
  - Learning curve

Scripting Module …

- Common languages used for scripting:
  - Python
    - http://www.python.org
  - Lua
    - http://www.lua.org
  - GameMonkey
    - http://www.somedude.net/gamemonkey
  - AngelScript
    - http://www.angelcode.com/angelscript

Animation

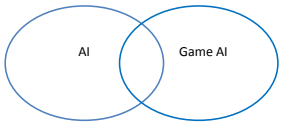
- Data structures are provided to:
  - Manage and support hierarchical rigging
  - Animate the scripts stored from mocap and animations hard coded by the animators
- The animation system needs to blend between the various canned animations and the physics interactions

AI Module

- Provides finite state automata and a behaviour model for the agents in the game to behave
- Also provides facilities for learned behaviours – Neural Networks
- Group behaviours (swarming), predator (seek), prey (flee) behaviours
- Path finding is another included component for Non-Player Character (NPCs) to find their way around a game environment

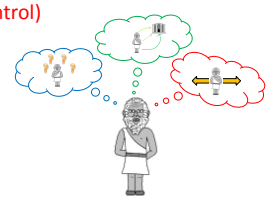
Game AI

- Different from conventional 'academic' AI
- Behaviour more important than intelligence
- Illusion of intelligence – not real intelligence



Game AI .

- Usually restricted to:
  - Decision making
  - Path finding (planning)
  - Steering (motion control)



Crowd Simulation

- Population of virtual environments
- Add realism to scenes that might otherwise look desolate

British & Colonial armies in "Empire Total War" (Creative Assembly)

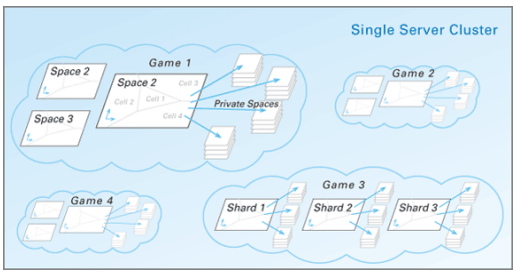
Crowd Simulation Example

- A battle in Empire Total War with over 30,000 units:
  - https://www.youtube.com/watch?v=v-xL_ITURB4

Networking

- Software to support network connections to other clients on a network or to servers
- Used to maintain verified game state on any machine connected to the server
- Sophisticated caching software is used to make sure every node is made aware of the present state of the game

Networking Example



http://bigworldtech.com/en/technology/server_en.php/

Scene Graph

- Tree data structure used to store the content
- Is operated upon by the Artificial Intelligence and Physics modules to update its state
- Fed to Graphics Module to be rendered into the framebuffer (screen memory)

User Interface

- Rather simple
  - But becoming more complex nowadays
- Monitors input devices and buffers any data received
- Displays menus and online help
  - Can nowadays be pretty complex
- Should be reusable, especially as a part of a game engine

Game Loop

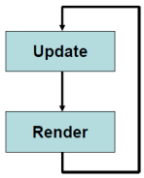
- A game is a real-time interactive application
- Three tasks that run concurrently:
  - Re-compute the state of the world
  - The player interacts with the world
  - The resulting state must be presented to the user
    - Graphics, sound, etc.
- Limitations of real-world technology
  - 1-2 processors with limited memory and speed

Game Loop .

- While user input not exit:
  - Update Scene-Graph with network cache
  - Update Scene-Graph via AI
  - Update Scene-Graph via Physics and Animation
  - Render Scene-Graph to Screen via Graphics System
- Endwhile

Game Loop: 1st Try

- Design update/render process in a single loop (coupled approach):

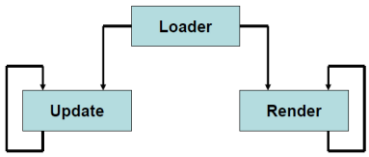


Advantages and Disadvantages

- Advantages of the coupled approach:
  - Both routines are given equal importance
  - Logic and presentation are fully coupled
- Disadvantages:
  - Variation in complexity in one of the two routines influences the other one
  - No control over how often a routine is updated

Game Loop: 2nd Try

- Design update process using two threads:



Advantages and Disadvantages

- Advantages of the multi-threaded approach:
  - Both update and render loops run at their own frame rate
- Disadvantages:
  - Not all machines are that good at handling threads
    - Precise timing problems
  - Synchronization issues
    - Two threads accessing the same data

Game Engine is Real-time

- All of this has to be updated at ~30 frames per second for a game
- Thus entire loop must run in 1/30th second or less for high quality animation rates
- Thus a very tight real-time system
- Thus programmer skills MUST be at a high level

Game Engine Developer

- Programs the core of the game engine
  - Programs one or multiple modules in the engine
- Tends to be a specialist in one area
  - i.e. Graphics component
- Usually the best programmers around
  - Needs a lot of experience

Typical Roles

- Engine Development
- Engine Research and Development
- Cross Platform Development
- Bug fixing with Quality Assurance (QA)
- Work with a lead developer in a development team
- Work with technical director who oversees a number of projects
- Work with animators and designers to negotiate technical requirements for the game

Skills Required

- High level Software Engineering skills – specification, testing, documentation skills
- Deep understanding of basic computer science algorithms – hashing, data structures, pointer arithmetic, code optimisation (necessary)
- C++ a necessity
- Usual analytical skills required from general ICT programming for specification/understanding of program design
- Problem solving skills – creative solutions and or bug fixing

Skills Required .

- Mathematics, Physics and AI skills very desirable
- Multiprocessor programming – concurrant programming of multiple threads (mini programs) - for the new multicore Xbox, PC and PS3s
- Do lots of development of your own mods - good
- Development of deep technical skills via creation of your own engine components
- People skills – yep, that again

Development Tools

- Interactive Development Environments (IDE)
  - Visual Studio, Eclipse, etc
- Sophisticated environments for the development of software:
  - Intelligent Editors
    - Think word processor for source code
  - Debuggers
    - Allows you to watch the code execute to find problems (bugs)
  - Profilers
    - Enables analysis of the efficiency of developed code

Development Tools .

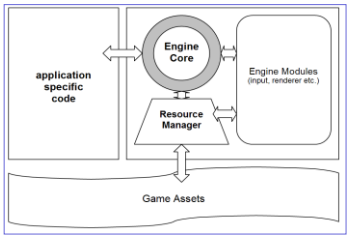
- Application Programming Interfaces (APIs)
  - Libraries of useful code to use within into your source code
  - DirectX – graphics/audio
  - OpenGL – graphics
  - Open Audio – audio
- Middleware – useful software APIs that facilitate various smaller tasks in games (goes between other components)
  - Physics, Data Processing, Networking, AI, User Interfaces

Console/Handheld Development Tools

- Console/Handheld Development Kits
- To develop for a console or handheld device
  - i.e. X360, PS3, DS, PSP, etc
- Specially manufactured hardware is used to communicate via a network link
- Remotely debug software on device
- Consoles only run signed code, so need dev. kit hardware for unsigned code
- Code is then given signature which runs on normal consoles

Game Engine Organisation

- "game code" – accesses core
- core – delegates tasks to sub-systems

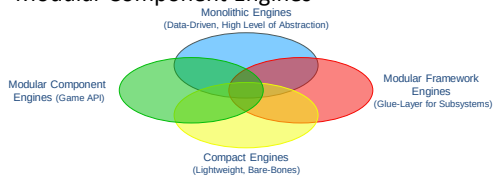


Engine Classification

- Game engines can be classified by:
  - Architecture (structure of system design)
    - Mode of access to features, ranging from indirect access through different abstractions, to direct access through individual function calls
  - Primary use (type/genre of game)
    - Available engines are often targeted at specific game genres, such as real-time-strategy (RTS) or first-person-shooter (FPS) games
  - Completeness (features and toolset)
    - The extent to which a game engine provides a one-stop solution to the development of computer games

Engine Architecture Categories

- Monolithic Engines
- Compact Engines
- Modular Framework Engines
- Modular Component Engines



Monolithic Game Engines

- A "Monolithic Game Engine" [Bishop et al. 1998] uses a high level of abstraction
- All of a game's components that are unique to the game are stored in external game assets
- These game engines are purely data-driven systems that are a 'player' for games, similar in the way that 'media player' programs are used for playing back audio or video data [BinSubaih et al. 2007]

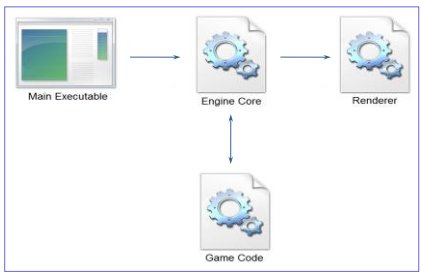
Compact Game Engines

- Compact Game Engines are relatively lightweight engines with limited features and an architecture of low complexity
- Due to their limitations they usually require the addition of additional middleware components to provide a more complete feature set
- Compact engines are simpler than modular engines and most of the early game engines fall into this category

Modular Framework Engines

- Modular Framework Engine are engines whose design has been separated out into a number of distinct modules that provide sub-systems to the engine
  - Usually provide different layers of access, including a high-level abstraction layer that will allow game construction without the need to explicitly access the engine's sub-systems (black-box)
  - Allow low-level access to sub-systems to be requested from the high-level layer to provide developers with additional control of the behaviour of the engine

Example: Modular Framework Engine - id Tech 2



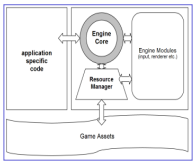
Described by [Arvesen 2003]

Modular Component Engines

- Modular Component Engines provide little more than a complex game development API
  - Contains all of the components that are necessary to build games, while all of the game logic, including the game loop, is kept separate from the engine [Franke 2005]
    - i.e. the simulation aspects are separated out and the engine is purely a means of providing input and output to the player

Engine Subsystems

- Components (outside the core)
  - That make up a game engine
- Common components:
  - Graphical output (rendering) component
  - A sound output component
  - A user input component

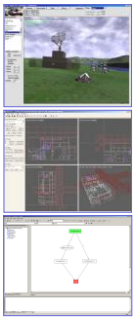


Subsystem Integration

- Creating control & data-flow connections between different engine subsystems
  - May require data to be transformed / customised for different subsystems
- Achievable through different methods (can be mixed)
  - Statically linked objects/libraries
  - Shared objects (loaded at startup)
  - Shared objects (managed dynamically)

Game Middleware

- Different from middleware in software engineering (integration software)
- Robust/reliable 3rd party software components used in own product
- Ranges from complete game engine solutions to parts of engine subsystems
- (Usually) includes service
- Often includes tools to simplify content creation for the software



Build Middleware

- Own technology = own IP
- Creates reusable core technology for later projects
- No code integration problems
- Up-front costs are reduced

Buy Middleware

- Focus on game (content) creation
- Includes service
  - i.e. developer can help with middleware integration
- No need for “core technology group”
- Development time is reduced

Middleware Usage

- Effectively outsourcing of R&D
- Almost all games use some kind of middleware
- In 2006 middleware accounted for 26% of the game industry’s spending
- One of the largest markets is middleware for mobile (phone etc.) game development

Middleware Pros

- Provides a generic solution
  - That may be genre specific
- Often cheaper than the cost of development
- Saves development time
- Generally well tested and stable

Middleware Cons

- Provides a generic solution
- Rarely works "out of the box"
  - Must be adapted
- Requires time consuming integration with the game
- Creates dependency on middleware developer

Good Middleware

- There is no good or bad middleware:
  - Middleware fits your requirements or it does not
- Some useful pointers – good middleware:
  - Allows the use of custom memory allocators
  - Allows the use of custom I/O functions
  - Is extensible (at a reasonably low level)
  - Has no external dependencies (avoiding symbolic conflicts)
  - Is thread-safe (allowing concurrency) and stable
  - Includes source code

[Wilson 2008]

Evaluating Middleware

- Evaluate performance
  - Against real-world scenarios
- Evaluate the learning curve
  - Hard to learn using middleware?
- Evaluate extensibility
- Evaluate technology
- Source code included?
- Evaluate support service

[Macris 2003]

Useful Components: STL

- STL can be very fast …
  - If used properly i.e. choice of correct containers, etc.
- Works well for the general case
  - Possible to code faster routines for special/specific cases if on is an expert programmer
  - Advice: Use it rather than trying to code your own basic classes
  - Afterwards, find time critical areas of code and see if they can be optimised separately
- Supposed to be standard, but differing implementations available…
  - STLport, Microsoft STL, SGI STL, etc.
  - So be careful!

Useful Components: SDL

- Simple Directmedia Layer
- Cross-platform multimedia library provides low-level access to:
  - Audio, Keyboard, Mouse, Joystick, 3D hardware via OpenGL, 2D video frame buffer, sounds, CD-ROM audio
  - Threads, timers
- Written in C, works in C++, bindings to many languages
  - Including C#, Python, Java, Lua, Perl, PHP…
- Free to use, even in commercial programs, as long as dynamic library version is linked to (at least up to v1.2)
  - LGPL license

http://www.libsdl.org/

Useful Components: AssImp

- Open Asset Import Library
- Cross-platform scene/model loading library provides functionality for:
  - Loading objects (meshes)
  - Processing/optimising meshes
  - Loading animation data
  - Exporting scenes/models (latest SVN version)
- Written in C++, bindings to many languages
- Free to use, even in commercial programs
  - BSD-style license

http://assimp.sourceforge.net/

Useful Components: OGRE

- Object-oriented Graphics Rendering Engine http://www.ogre3d.org/
  - OGRE is primarily a graphics engine
  - Does not concentrate on sound, AI, networking, collision, physics, although these are often available as add-ons
  - Can be made into a game engine with lots of work
- Lots of features
- Ambient occlusion, parallax mapping, soft shadows, etc

Useful Components: FMOD

- Music and sound effects system http://www.fmod.org
- Library and toolkit for creation and playback of interactive audio
- Cross-platform, supporting many 'next gen' consoles
- Regular releases
- FMOD Ex provides low-level API and data-driven API
  - Includes a suite of effects, such as echo, chorus, reverb, etc.
  - Supports numerous file formats: wav, mp3, midi, XMA, mod
  - Can play audio files with up to 16 channels
- Virtual voices enables thousands of sounds to be played at once on limited hardware
  - Voice management using priorities and 3D distance metrics

Useful Components: ODE

- Open Dynamics Engine http://www.ode.org
- High performance library for simulating rigid body dynamics
- Relatively easy to use C/C++ API
- Advanced joint types and integrated collision detection
- Appropriate for simulating vehicles, creatures and interactions between objects
- BSD license: free for use in commercial products

Useful Components: Lua

- Lua "extensible extension language" http://www.lua.org
- Embeddable scripting language compiler & virtual machine with a very small footprint.
- Very easy to use C/C++ API
- Very easy to use syntax
- Used widely in game development
- MIT license: free for use in commercial products

Software Engineering

- Definition from Collins Dictionary of Personal Computing:
  - The discipline of providing software of a high standard
  - This includes analysis of the problem which the software is designed to solve, searching for methods that provide the minimum use of memory and the lowest running time at minimum expense to the user, programming and marketing
  - This is engineering in the sense that it involves the traditional engineering problems of reconciling conflicting objectives and working with a view to acceptance by the ultimate user
- Game development greatly benefits from adopting software engineering methods & best practices!!!

Software Prototyping

- Prototyping is an iterative software design method, useful for game development
  - Prototyping employs a top-down approach of stepwise refinement
  - Working parts of the application are created for testing of ideas and concepts
  - In successive iterations the functionality of early prototypes may be improved or replaced by rewrites from scratch

Revision Control

- Tools that allow versioning of software in development
  - Several developers can work simultaneously
  - Old versions are stored, allowing development to "roll back" if problems arise
- Popular examples:
  - CVS (Concurrent Versions System)
  - SVN (Subversion)
  - GIT

Reuse & Refactoring

- Never design code for reuse [Norneby and Olsson 2009]
  - Design code that you need now, not code that you might need in the future
  - A good system will automatically have many reusable features, many of which will not have been anticipated
- Factor out common code elements (reused from a different project) into an external library
- Control change – ensure that any code restructuring that happens does not change the functionality of existing code (unless desired)

Subsystem Integration

- Creating control & data-flow connections between different engine subsystems
  - May require data to be transformed / customised for different subsystems
- Achievable through different methods (can be mixed)
  - Statically linked objects/libraries
  - Shared objects (loaded at startup)
  - Shared objects (managed dynamically)

Middleware Integration Issues

- Language clash – middleware may require C-style callbacks, whereas the engine uses C++
  - No "pretty" solutions – can be solved using bad C++ programming style
    - i.e. wrapper functions, global variables
- Middleware may require existing data in a different format
  - Data replication in correct formats – synchronisation issues & excessive memory usage
  - Conversion functions – execution overheads
  - No ideal solution

Callbacks

- Programming interface for event-driven input
- Define a callback function for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example:
  - glutMouseFunc(mymouse)

Rendering

- Goal:
  - Transform computer models into images
  - May or may not be photo-realistic
- Methods:
  - Interactive rendering
  - Offline rendering

Interactive vs Offline Rendering

- Interactive rendering
  - Fast, but limited quality
  - Roughly follows a fixed patterns of operations
    - Rendering pipeline
- Offline rendering
  - Ray tracing
  - Radiosity
  - Global illumination

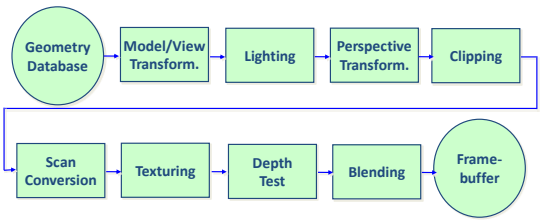
Rendering Tasks

- Tasks that must be performed (no particular order):
  - Project all 3D geometry onto the image plane
    - Geometric transformations
  - Determine which primitives of primitives are visible
    - Hidden surface removal
  - Determine which pixels a geometric primitive covers
    - Scan conversion
  - Compute the color of every visible surface point
    - Lighting, shading, texture mapping

What is the Rendering Pipeline?

- What is the pipeline?
  - Abstract model for sequence of operations to transform geometric model into digital image
  - Abstraction of the way graphics hardware works
  - Underlying model for application programming interfaces (APIs) that allow programming of graphics hardware
    - OpenGL
    - Direct 3D
- Actual implementation details of rendering pipeline will vary

The Rending Pipeline



Pipeline Advantages

- Modularity: logical separation of different components
- Easy to parallelize
  - Earlier stages can already work on new data while later stages still work with previous data
  - Similar to pipelining in modern CPUs
  - But much more aggressive parallelization possible (special purpose hardware!)
  - Important for hardware implementations
- Only local knowledge of the scene is necessary

Pipeline Disadvantages

- Limited flexibility
- Some algorithms would require different ordering of pipeline stages
  - Hard to achieve while still preserving compatibility
- Only local knowledge of scene is available
  - Shadows, global illumination difficult

OpenGL

- Started in 1989 by Kurt Akeley
  - Based on IRIS_GL by SGI
- API to graphics hardware
- Designed to exploit hardware optimized for display and manipulation of 3D graphics
- Implemented on many different platforms
- Low level, powerful flexible
- Pipeline processing
  - Set state as needed

Developer-Driven Advantages

- Industry standard
  - Vendor-neutral, multiplatform graphics standard
- Stable
- Reliable and portable
- Evolving
- Scalable
  - Consumer electronics to PCs, workstations, and supercomputers
- Very easy to use
- Well-documented

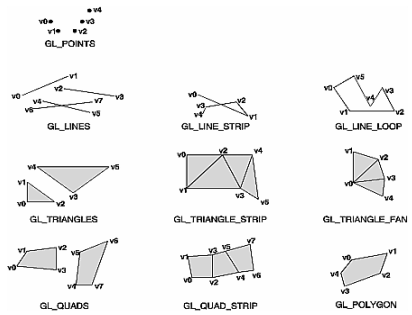
OpenGL Graphics State

- Set the state once, remains until overwritten
- glColor3f(1.0, 1.0, 0.0)
  - Set color to yellow
- glSetClearColor(0.0, 0.0, 0.2)
  - Dark blue background
- glEnable(LIGHT0)
  - Turn on a light
- glEnable(GL_DEPTH_TEST)
  - Hidden surface

Geometry Pipeline

- Tell it how to interpret geometry
  - glBegin(<mode of geometric primitives>)
  - mode = GL_TRIANGLE, GL_POLYGON, etc.
- Feed it vertices
  - glVertex3f(-1.0, 0.0, -1.0)
  - glVertex3f(1.0, 0.0, -1.0)
  - glVertex3f(0.0, 1.0, -1.0)
- Tell it you're done
  - glEnd()

OpenGL Geometric Primitives



Simple Tasks for OpenGL

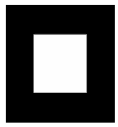
- The basic structure of a useful OpenGL program can be very simple
- The main tasks include:
  - Initialisation of certain states that control how OpenGL rendering is done
  - Specification of the objects/scenes to be rendered

OpenGL Includes

- For all OpenGL applications always include:
  - #include <GL/gl.h>
  - #include <GL/glu.h>
- When using GLUT (see next slides) include:
  - #include <GL/glut.h>

A Code Sample

```
void display()
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glBegin(GL_POLYGON);
         glVertex3f(0.25, 0.25, 0.0);
         glVertex3f(0.75, 0.25, 0.0);
         glVertex3f(0.75, 0.75, 0.0);
         glVertex3f(0.25, 0.75, 0.0);
   glEnd();
}
```



Lighting in OpenGL

- Steps required to add lighting to your scene
  - Define normal vectors for each vertex of all the objects
    - These normals determine the orientation of the object relative to the light sources
  - Create, select, and position one or more light sources
  - Create and select a lighting model, which defines the level of global ambient light and the effective location of the viewpoint
    - For the purposes of lighting calculations
  - Define material properties for the objects in the scene

Create, Position, and Enable One or More Light Sources

- A white light source is specified by the glLightfv() call
  - If you want a differently colored light, use glLight*() to indicate this
- You can include at least eight different light sources in your scene of various colors
- The default color of these other lights is black
- Remember that each light source adds significantly to the calculations needed to render the scene
  - Performance is affected by the number of lights in the scene

Select a Lighting Model

- The glLightModel*() command describes the parameters of a lighting model
- The lighting model also defines whether:
  - The viewer of the scene should be considered to be an infinite distance away or local to the scene
- Lighting calculations should be performed differently for the front and back surfaces of objects in the scene

Define Material Properties for the Objects in the Scene

- An object's material properties determine how it reflects light
- Specify material properties so that an object has a certain desired appearance is an art
- You can specify a material's ambient, diffuse, and specular colors and how shiny it is

Defining Colors and Position for a Light Source Example

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Position and Attenuation Examples

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_POSITION, light_position);

glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION,
   2.0);

glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);

glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION,
   0.5);
```

Two-sided Lighting

- Lighting calculations are performed for all polygons
  - Whether they're front-facing or back-facing
  - Usually set up lighting conditions with front-facing polygons
  - However, the back-facing ones typically aren't correctly illuminated
- When you turn on two-sided lighting with
  - glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
- OpenGL reverses the surface normals for back-facing polygons
  - Typically, this means that the surface normals of visible back- and front-facing polygons face the viewer, rather than pointing away
- As a result, all polygons are illuminated correctly

Emission

- By specifying an RGBA color for GL_EMISSION, can make an object appear to be giving off light of that color
- Since most real-world objects (except lights) don't emit light, use this feature mostly to simulate lamps and other light sources in a scene

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

- Need to create a light source and position it at the same location as the sphere to create that effect

OpenGL Lighting Hints

- Lighting in OpenGL is not so easy!
- Must get light sources, materials, and surface normals all correct
- If any one of them is wrong, it doesn't work at all
- As always, the key to successful learning is to keep things simple and change only one value at a time
- Don't go crazy with lights
  - Most scenes can be rendered quite comfortably with just one


OpenGL Lighting Hints .

- The light values for ambient, diffuse, etc are RGB colors
  - Use only white lights (R=G=B)
  - The fourth (alpha) value should always be 1
- Ambient light can be either specified globally or on your primary light source only
  - Intensity should be in the 0.1 to 0.3 range
  - Never have multiple ambient lights
- Diffuse lighting is what photographers call 'soft' lighting
  - It should always be present, intensity 1


OpenGL Lighting Hints ..

- Specular light is 'hard' lighting that creates highlights, sparkles, etc
  - Your primary light source always has specular intensity 1
  - If you need secondary lights as the equivalent of a photographers backlighting or extra flash, set the specular to 0 on those
- Lights have a position (x, y, z,1) or a direction (x, y, z, 0)
  - Directional lights are easier to work with
- For a directional light the coordinates are a vector (the direction from which the light comes)
  - Try to use only the values 0 or 1 at each of (x, y, z)
  - The vectors (0, 1, 0) and (0, 0, 1) are good values for directional lighting


OpenGL Materials Hints

- Can make something appear yellow by either coloring it yellow and shining a white light on it, or coloring it white and shining a yellow light on it
  - The first is easier, so lights are set up once and rarely altered
- Materials are where most of your design will be done
- Ambient and diffuse material values should be identical
  - OpenGL allows you to specify both at the same time for this reason


OpenGL Materials Hints .

- Very roughly, all materials are either:
  - Matte: with high diffuse color and low specular
  - Plastic: with high diffuse color and white specular
  - Metal: with low diffuse and high specular color
- When something is described as being a particular color, that means the diffuse color for matte and plastic surfaces, the specular for metals


OpenGL Materials Hints ..

- OpenGL has the capability to set material values from the current vertex color values through use of **glColorMaterial**
  - While this is very useful to experienced programmers, it is sometimes tricky
- Using glColorMaterial does not mean you don't have to set up materials for your objects
  - It just adds one more thing that can go wrong


OpenGL Surface Normals Hints

- Surface normals are essential for lighting
  - Always irritating to calculate!
- Where possible, use the glu_ and glut_ primitives such as spheres and cylinders
  - They calculate the surface normals for you
  - Good for testing your own lights and materials
    - Can't get the normals wrong


OpenGL Surface Normals Hints .

- A surface normal is a vector perpendicular to the polygon (triangle, etc) being lit
  - It is in local coordinates before any transformations
  - If you rotate or translate say, a cylinder, the surface normals get transformed with it
- When learning to calculate normals, start with a cube that you construct yourself
  - The top face has surface normal (0, 1, 0), straight up, and all the others are equally easy


OpenGL Surface Normals Hints ..

- The surface normal has to be a normal or unit length vector
  - Otherwise the lighting calculations won't work
- If you use glScale, surface normals may no longer be correct because the scaling will shorten or lengthen the vector
  - At startup, write:
    - glEnable(GL_NORMALIZE);
      - OpenGL will re-normalise all your surface normals


GLUT: OpenGL Utility Toolkit

- Developed by Mark Kilgard (also from SGI)
- Simple, portable window manager
  - Opening windows
    - Handling graphics contexts
  - Handling input with callbacks
    - Keyboard, mouse, window reshape events
  - Timing
    - Idle processing, idle events
  - Designed for small-medium size applications
  - Distributed as binaries


Create a Window in GLUT

```
int main(int argc, char **argv)
{
   glutInit( &argc, argv );
   glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE
   | GLUT_DEPTH);
   glutInitWindowSize( 640, 480 );
   glutCreateWindow( "openGLDemo" );
   glutDisplayFunc( DrawWorld );
   glutIdleFunc(Idle);
   glClearColor( 1,1,1 );
   glutMainLoop();
   return 0;
}
```


Event-Driven Programming

- Main loop not under your control
  - vs. batch mode where you control the flow
- Control flow through event callbacks
  - Redraw the window now
  - Key was pressed
  - Mouse moved
- Callback functions called from main loop when events occur
  - Mouse/keyboard state setting vs. redrawing


GLUT Callback Functions

```
// you supply these kind of functions
   void reshape(int w, int h);
   void keyboard(unsigned char key, int x, int y);
   void mouse(int but, int state, int x, int y);
   void idle();
   void display();

 // and register them with glut
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMouseFunc(mouse);
   glutIdleFunc(idle);
   glutDisplayFunc(display);
```


More GLUT Callback Functions

- void glutDisplayFunc (void (*func)(void));
- void glutKeyboardFunc
  (void (*func)(unsigned char key, int x, int y));
- void glutIdleFunc (void (*func)());
- void glutReshapeFunc
  (void (*func)(int width, int height));


GLUT Callbacks

- GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)
  - glutDisplayFunc
  - glutMouseFunc
  - glutReshapeFunc
  - glutKeyboardFunc
  - glutIdleFunc
  - glutMotionFunc
  - glutPassiveMotionFunc


GLUT Event Loop

- Recall that the last line in main.c for a program using GLUT must be:
  - glutMainLoop();
  - Which puts the program in an infinite event loop
- In each pass through the event loop, GLUT
  - Looks at the events in the queue
  - For each event in the queue, GLUT executes the appropriate callback function if one is defined
  - If no callback is defined for the event, the event is ignored


The display Callback

- The display callback is executed whenever GLUT determines that the window should be refreshed, for example
  - When the window is first opened
  - When the window is reshaped
  - When a window is exposed
  - When the user program decides it wants to change the display


The display Callback .

- In main.c
  - glutDisplayFunc(mydisplay) identifies the function to be executed
  - Every GLUT program must have a display callback


Posting Redisplays

- Many events may invoke the display callback function
  - Can lead to multiple executions of the display callback on a single pass through the event loop
- Can avoid this problem by instead using
  - glutPostRedisplay();
    - Which sets a flag
- GLUT checks to see if the flag is set at the end of the event loop
  - If set then the display callback function is executed

Animating a Display

- When we redraw the display through the display callback, we usually start by clearing the window
  - glClear()
  - Then draw the altered display
- Problem: the drawing of information in the frame buffer is decoupled from the display of its contents
  - Graphics systems use dual ported memory
- Hence we can see partially drawn display

Double Buffering

- Instead of one color buffer, we use two
  - Front Buffer: one that is displayed but not written to
  - Back Buffer: one that is written to but not displayed
- Program then requests a double buffer in main.c
  - glutInitDisplayMode(GL_RGB | GL_DOUBLE)
  - At the end of the display callback buffers are swapped

```
void mydisplay()
{
   glClear(GL_COLOR_BUFFER_BIT|….)
   /* draw graphics here */
   glutSwapBuffers()
}
```

Using the idle Callback

- The idle callback is executed whenever there are no events in the event queue
  - glutIdleFunc(myidle)
  - Useful for animations

```
void myidle() {
    /* change something */
    t += dt
    glutPostRedisplay();
    }

void mydisplay() {
    glClear();
    /* draw something that depends on t */
    glutSwapBuffers();
    }
```

Using Globals

- The form of all GLUT callbacks is fixed
  - void mydisplay()
  - void mymouse(GLint button, GLint state, GLint x, GLint y)
- Must use *globals* to pass information to callbacks

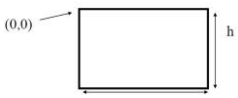
```
float t; // global variable
void mydisplay()
{
     // draw something that depends on t
}
```

The mouse callback

- glutMouseFunc(mymouse)
  - void mymouse(GLint button, GLint state, GLint x, GLint y)
- Returns
  - Which button (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON) caused event
  - State of that button (GLUT_UP, GLUT_DOWN)
  - Position in window

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert y coordinate returned by callback by height of window
  - y = h – y;



Obtaining the Window Size

- To invert the *y* position we need the window height
  - Height can change during program execution
  - Track with a global variable
  - New height returned to reshape callback that we will look at in detail soon
- Can also use query functions
  - glGetIntv
  - glGetFloatv
    - To obtain any value that is part of the state

Using the Mouse Position

- To draw a small square at the location of the mouse each time the left mouse button is clicked
  - This example does not use the display callback but one is required by GLUT
  - We can use the empty display callback function
  - mydisplay(){}

Drawing Squares at Cursor Location

```
void mymouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_RIGHT_BUTTON &&
   state==GLUT_DOWN)
        exit(0);                    // exits the
   application
   if(btn==GLUT_LEFT_BUTTON &&
   state==GLUT_DOWN)
        drawSquare(x, y);           // draw a square
}
```

Drawing Squares at Cursor Location .

```
void drawSquare(int x, int y){
   y=w-y; // invert y position
   glColor3ub( (char) rand()%256, (char) rand
   )%256, (char) rand()%256); // random color
   glBegin(GL_POLYGON);
        glVertex2f(x+size, y+size);
        glVertex2f(x-size, y+size);
        glVertex2f(x-size, y-size);
        glVertex2f(x+size, y-size);
   glEnd();
}
```

Using the motion callback

- We can draw squares (or anything else) continuously as long as a mouse button is depressed by using the motion callback
  - glutMotionFunc(drawSquare)
- We can draw squares without depressing a button using the passive motion callback
  - glutPassiveMotionFunc(drawSquare)

Using the Keyboard

```
glutKeyboardFunc(mykey)
void mykey(unsigned char key, int x, int y)
```

- Returns ASCII code of key depressed and mouse location

```
void mykey()
{
   if(key == 'Q' | key == 'q')
   exit(0);
}
```

Special and Modifier Keys

- GLUT defines the special keys in glut.h
  - Function key 1: GLUT_KEY_F1
  - Up arrow key: GLUT_KEY_UP
    - if(key == 'GLUT_KEY_F1' ……
- glutGetModifiers() allows emulation of three-button mouse with one- or two-button mice by checking if one of the modifiers is depressed
  - GLUT_ACTIVE_SHIFT
  - GLUT_ACTIVE_CTRL
  - GLUT_ACTIVE_ALT

Reshaping the Window

- We can reshape and resize the OpenGL display window by pulling the corner of the window
- What happens to the display?
  - Must redraw from application
- Two possibilities
  - Display part of world
  - Display whole world but force to fit in new window
    - Can alter aspect ratio

The reshape Callback

glutReshapeFunc(myreshape)
void myreshape(int w, int h)

- Returns width and height of new window (in pixels)
- A redisplay is posted automatically at end of execution of the callback
- GLUT has a default reshape callback but you probably want to define your own
- The reshape callback is a good place to put viewing functions because it is invoked when the window is first opened

Example Reshape

- This reshape preserves shapes by making the viewport and world window have the same aspect ratio

```
void myReshape(int w, int h){
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION); // switch matrix mode
   glLoadIdentity();
   if (w <= h)
   gluOrtho2D(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 *
   (GLfloat) h /        (GLfloat) w);
   else gluOrtho2D(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 *
   (GLfloat) w /        (GLfloat) h, -2.0, 2.0);
   glMatrixMode(GL_MODELVIEW); /* return to modelview
   mode */
}
```

Toolkits and Widgets

- Most window systems provide a toolkit or library of functions for building user interfaces that use special types of windows called widgets
- Widget sets include tools such as:
  - Menus
  - Slidebars
  - Dials
  - Input boxes
- But toolkits tend to be platform dependent
- GLUT provides a few widgets including menus

GLUT Menus

- GLUT supports pop-up menus
  - A menu can have submenus
- Three steps
  - Define entries for the menu
  - Define action for each menu item
    - Action carried out if entry selected
  - Attach menu to a mouse button

Defining A Simple GLUT Menu

```
menu_id = glutCreateMenu(mymenu);
glutAddmenuEntry("clear Screen", 1);
glutAddMenuEntry("exit", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

GLUT Menu Actions

- Menu callback

```
void mymenu(int id)
{
   if(id == 1) glClear();
   if(id == 2) exit(0);
}
```

- Note each menu has an *id* that is returned when it is created
- Add submenus by:
  - glutAddSubMenu(char *submenu_name, submenu id)

Timers

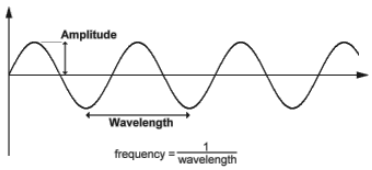
- On modern graphics processors may need to slow down rendering or get a blur
- Options
  - Use OS timers
  - Lock buffer swap on graphics card to refresh rate
  - Use GLUT timer
- glutTimerFunc(int delay, void(*timer_func)(int), int value);
  - Delay the event loop for delay seconds

What is Sound?

- Sound is caused when a vibrating object creates waves of pressure
- These waves travel through the air (or other medium)
  - Speed of sound in air ~340 m/s
  - Water 1480m/s, Steel 4000m/s
- The waves radiate outwards from their source in a spherical manner
- The pressure waves diminish with distance

Sound waves

- Sound waves are often simplified to a description in terms of sinusoidal plane waves

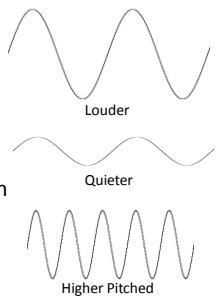


Properties of sound wave

- Frequency, or its inverse, the period
- Wavelength
- Wavenumber
- Amplitude
- Sound pressure
- Sound intensity
- Speed of sound
- Direction

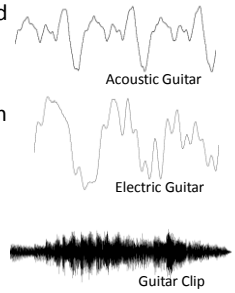
How do we Perceive Sound?

- Our ears are very sensitive to such pressure waves
- The volume of a sound is determined by the amount of pressure variation
  - Amplitude of the waves
- The pitch of a sound (how high or low it is) is driven by how quickly the pressure varies
  - Frequency of the waves



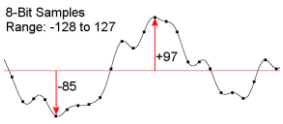
Waveforms

- Draw waveforms to represent the pressure variation created at the source
- Simple waveforms make very simple sounds
- Real-world sounds have much more complex waveforms
  - And waveform is densely packed
- Computers cannot store these analogue waveforms
  - Use a digital representation instead



Sampling Sounds

- A computer stores a sound by sampling the waveform
- A sample is just the measurement of a waveform at a given time



Sampling Sounds .

- Samples taken at a fixed sample frequency (rate)
  - e.g. CDs sampled at 44100Hz : 44100 samples / second
- Each sample is a value representing the amplitude (pressure) of the waveform at that point
- Samples are usually integers
  - Hence they have a max/min value
    - Not realistic

Digital Sound Quality

- Higher sample rates result in a better approximation to the original sound and hence a better quality sound
  - Nyquist criterion: sample rate must be greater than 2 x required max frequency in the sound
  - Typical sample frequencies: 11,000->48,000Hz
- The bit-depth of the samples is the number of bits used for each sample and determines the accuracy of each sample
  - E.g. An 8-bit, 16-bit or even floating point value for each sample
- A higher bit-depth will also improve digital sound quality
  - But improvement is less than the effect of increasing sample rate

Stereo Sound

- Each ear receives a different waveform from a single sound
- If the sound is off-centre then the sound waves will reach one ear sooner (and louder)
- Also, our head is an obstacle for the pressure waves and they distort when passing by
  - So the position of the head relative to the sound affects the received waveforms at each ear
- We can model this difference by recording stereo sound: 2 waveforms, one for each ear
  - Greatly improves sound quality
  - But pre-recorded sound is statically positioned

3D Sound

- Can dynamically model the different sounds received by our ears:
  - We first consider the 3D position of the sound source
  - Then calculate how the sound emitted will be perceived at each ear of the observer
- Need to take account of:
  - The 3D position of the sound relative to the observer
  - The head position (HRTF – head related transfer functions are used here)
  - Any obstacles in the way (e.g. walls)
- These calculations can be performed in hardware
- Can then model dynamic sound in our 3D scenes

Doppler Effect

- We have considered the relative 3D position of a sound source
- But the relative velocity also has an effect on the perceived sound:
  - Sounds moving towards an observer are higher in pitch
  - Those moving away are lower in pitch
- The waves are compressed / stretched by the movement:
- This should also be modelled to create a compelling 3D sound
- Also typically supported in hardware

Animation by Dr. Dan Russell, Kettering University

Other Special Effects

- There is other special processing we may perform on a waveform:
  - Echo, reverberation, distortion, etc.
  - Involve numeric processing of the samples
- Reverberation is most interesting for games
  - The reflection and damping of a sound against the walls and furniture of a environment
- We often apply an environmental reverberation to all sounds depending on the current scene

Basic Sound

Church

Bathroom

OpenAL

- OpenAL is a simple, effective audio API
  - Free, open source project
  - Supported by Creative Labs
  - Cross-platform, supports Windows, Macs, Linux
- Appropriate for games
  - Particularly good for 3D sound
  - Used in several titles
- Fairly easy to use
  - Similar principles to OpenGL

OpenAL Libraries

- OpenAL
  - #include <al\al.h>
  - #include <al\alu.h>
  - #include <al\alc.h>
- ALUT
  - #include <al\alut.h>

Init

- Similar to OpenGL
- In main()

```
//initialize OpenAL
alutInit(&argc, argv) ;
```

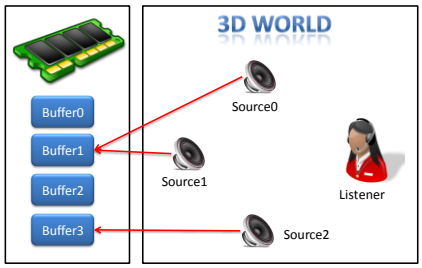
OpenAL Concepts

- OpenAL introduces three basic concepts:
  - Buffers:
    - A buffer holds sound data in memory
    - Creating a buffer doesn't play a sound
  - Sources:
    - An actual sound in the world.
    - Must be associated with a buffer
  - The Listener:
    - OpenAL always assumes there is a listener

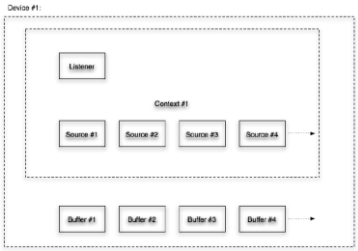
OpenAL Basics

- Buffers, sources & listener have properties:
  - Buffers:
    - Sample rate, bit depth, and other source data related properties
  - Sources:
    - Pitch, gain (volume), looping, position, velocity, etc.
  - The Listener
    - Position, orientation, velocity, master gain (volume)
  - Relative position of source / listener used to determine how to play sound out of speakers
  - Velocity is used to generate Doppler effect

OpenAL Architecture



Fundamental OpenAL Objects



OpenAL Programmer's Guide

Listener

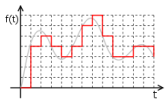
- For every context, there is automatically one Listener object

alListenerfv(AL_POSITION, listenerPos);
alListenerfv(AL_VELOCITY, listenerVel);
alListenerfv(AL_ORIENTATION, listenerOri);

Property	Data Type	Description
AL_GAIN	f, fv	"master gain" value should be positive
AL_POSITION	fv, 3f, iv, 3i	X, Y, Z position
AL_VELOCITY	fv, 3f, iv, 3i	velocity vector
AL_ORIENTATION	fv, iv	orientation expressed as "at" and "up" vectors

Buffer

- Each buffer generated by alGenBuffers has properties which can be retrieved



Property	Data Type	Description
AL_FREQUENCY	i, iv	frequency of buffer in Hz
AL_BITS	i, iv	bit depth of buffer
AL_CHANNELS	i, iv	number of channels in buffer > 1 is valid, but buffer won't be positioned when played
AL_SIZE	i, iv	size of buffer in bytes
AL_DATA	i, iv	original location where data was copied from generally useless, as was probably freed after buffer creation

Buffer Example

```
const int NUM_BUFFERS = 3;
ALuint    buffer[NUM_BUFFERS];
ALboolean al_bool;
ALsizei size,freq;
ALenum format;
ALvoid *data = NULL;
int ch;

    // Generate buffers, or no sound will be produced
    alGenBuffers(NUM_BUFFERS, buffer);

    if(alGetError() != AL_NO_ERROR) {
            printf("- Error creating buffers !!\n");
            exit(1);
    }
    alutLoadWAVFile("c.wav", &format ,&data, &size, &freq, &al_bool);
    alBufferData(buffer[0], format, data, size, freq);
    alutUnloadWAV(format, data, size, freq);
```

Source

- A source in OpenAL is exactly what it sounds like, a source of a sound in the world

Property	Data Type	Description
AL_PITCH	f, fv	pitch multiplier always positive
AL_GAIN	f, fv	source gain value should be positive
AL_MAX_DISTANCE	f, fv, i, iv	used with the Inverse Clamped Distance Model to set the distance where there will no longer be any attenuation of the source
AL_ROLLOFF_FACTOR	f, fv, i, iv	the rolloff rate for the source default is 1.0
AL_REFERENCE_DISTANCE	f, fv, i, iv	the distance under which the volume for the source would normally drop by half (before being influenced by rolloff factor or AL_MAX_DISTANCE)
AL_MIN_GAIN	f, fv	the minimum gain for this source
AL_MAX_GAIN	f, fv	the maximum gain for this source
AL_CONE_OUTER_GAIN	f, fv	the gain when outside the oriented cone
AL_CONE_INNER_ANGLE	f, fv, i, iv	the gain when inside the oriented cone
AL_CONE_OUTER_ANGLE	f, fv, i, iv	outer angle of the sound cone, in degrees default is 360
AL_POSITION	fv, 3f	X, Y, Z position
AL_VELOCITY	fv, 3f	velocity vector
AL_DIRECTION	fv, 3f, iv, 3i	direction vector
AL_SOURCE_RELATIVE	i, iv	determines if the positions are relative to the listener

Source Example

```
const int NUM_SOURCES = 3;
ALuint    source[NUM_SOURCES];
alGetError(); /* clear error */
    alGenSources(NUM_SOURCES, source);

    if(alGetError() != AL_NO_ERROR) {
            printf("- Error creating sources !!\n");
            exit(2);
    }

    alSourcef(source[0], AL_PITCH, 1.0f);
    alSourcef(source[0], AL_GAIN, 1.0f);
    alSourcefv(source[0], AL_POSITION, source0Pos);
    alSourcefv(source[0], AL_VELOCITY, source0Vel);
    alSourcei(source[0], AL_BUFFER, buffer[0]); //attach buffer
    alSourcei(source[0], AL_LOOPING, AL_TRUE);
```

Play and Stop

- Combine with keyboardfunc(), or some other way
  - alSourcePlay(source[0]);
  - alSourceStop(source[0]);
  - alSourcePause(source[0]);

Exit

```
In main()
// Setup an exit procedure.
atexit(KillALData);

void KillALData()
 {
alDeleteSources(NUM_SOURCES, source);
   alDeleteBuffers(NUM_BUFFERS, buffers);
alutExit();
 }
```

Other APIs

- DirectSound is part of the DirectX SDK
  - Deprecated now, may see in older projects
- XAudio2 is DirectSound's replacement
  - Cross-platform API for Windows and 360
  - Fully featured and effective, many tools/APIs Proprietary, not open source
  - Rather more complex than OpenAL
- FMOD Ex is a free API, widely used for games
  - Supports all platforms (Windows, Linux, PS3, Xbox)
  - Low-level and a little more complex than OpenAL

References

- Anderson, E.F., Engel, S., McLoughlin, L. and Comninos, P. (2008). The case for research in game engine architecture. In Proceedings of the ACM FuturePlay 2008 International Academic Conference on the Future of Game Design and Technology, pp. 228–231
- Arvesen, R. (2003). Quake II.NET. Whitepaper, Vertigo Software - http://www.vertigosoftware.com
- Binsubaih, A., Maddock, S. and Romano, D. (2007). A survey of 'game' portability. Tech. Rep. CS-07-05, University of Sheffield. Department of Computer Science.
- Bishop, L., Eberly, D., Whitted, T., Finch, M. and Shantz, M. (1998). Designing a PC game engine. IEEE Comput. Graph. Appl. 18(1), pp. 46–53
- Blow, J. (2004). Game Development: Harder than you think, ACM Queue 1(10), pp. 28–37
- Franke, S. (2005). Game Development Architecture. Tech. Rep. TR-CS-2005-01, Fachhochschule Augsburg, pp. 172-182

References .

- Franke, S. (2005). Game Development Architecture. Tech. Rep. TR-CS-2005-01, Fachhochschule Augsburg, pp. 172-182
- Zerbst, St, Düvel, O. and Anderson, E. (2003). 3D-Spieleprogrammierung. Markt + Technik
- Macris, A. (2003). Effective Middleware Evaluation. Game Developer 10(5)
- Wilson, K. (2008). Gamasutra - http://www.gamasutra.com/php-bin/news_index.php?story=20406
- Norneby, J. and Olsson, T. (2009). A New Attitude To Game Engineering: Embrace Change, Re-Use, Fun. Gamasutra.com
- Blow, J. (2004). Game Development: Harder than you think, ACM Queue 1(10), pp. 28–37.
- El Rhalibi, A., England, D. and Costa, S. (2005). %T Game Engineering for a Multiprocessor Architecture. In Changing Views: Worlds in Play - Proceedings of the 2005 Digital Games Research Association Conference.

Useful Links

- GLUT Libraries
  - http://www.xmission.com/~nate/glut.html
- GLUT Tutorials
  - http://www.lighthouse3d.com/opengl/glut/
  - http://www.opengl.org/code/category/C19
  - http://www.nullterminator.net/glut.html
  - http://www.zeuscmd.com/tutorials/glut/

Useful Links .

- http://devmaster.net/devdb/engines
- http://www.gamedev.net/page/resources/_/technical/game-programming/a-guide-to-starting-with-openal-r2008
- http://www.openal.org/creative-installers/
- http://www.openal.org/documentation/openal-1.1-specification.pdf

Questions