Rendering Engine Design

An introduction

Camilla Berglund
elmindreda@stacken.kth.se

http://www.elmindreda.org/opengl/

You need to know...

- Simple mathematics
  - Linear algebra, trigonometry, etc.
- Basic CG terminology and concepts
- Fundamentals of OpenGL
  - The previous lecture should be enough
  - Familiarity with Direct3D should also help
- C++ with (some) templates

Engine *n.*

During this lecture, an engine is...

Specifically a *rendering* engine, not a game, physics or combustion engine

Synonymous with renderer, unless otherwise stated

Implemented for real-time rendering on current or previous generation graphics hardware and APIs

What is covered

- Common concepts and design patterns
  - Rendering pipeline structure
  - Scene organisation and partitioning
  - (Some) resource management
- An overview of a simple renderer
- Pointers to other solutions
  - Open and closed source engines
  - Articles, books, websites, IRC channels

What isn't covered

- Every concievable solution
- Advanced techniques
  - ...for some arbitrary value of advanced
- Data creation and conversion
  - Including geometry, textures, shaders, animations, scripts, fonts, etc.
- Physics and collision systems
  - This isn't about simulation engines

Part 1

Crash course in modern OpenGL

The old graphics API

- Fixed-format vertices
- Immediate mode and array pointers
- Fixed set of specialised matrices
- Huge set of state variables
- Video memory was used for textures and the framebuffer
- Extensions added more states controlling tiny, static black boxes

The modern graphics API

- Vertex and element buffers
- Vertex and fragment pipelines programmable with high-level languages compiled by the driver
- Primitive level programmability coming
- Video memory also stores geometry, shader constants and other data
- Extensions improve the languages, extend size limits and format precision

The future graphics API

- Everything is either a buffer or a shader
- Shaders that create geometry, optionally saving results back to VRAM
- Shaders with stack pointer, scratch memory, integer operations, unlimited execution time, etc.
- Predicated rendering for occlusion
- Extensions... who knows?

Stop using...

- Immediate mode (period)
  - I.e. `glVertex`, `glNormal`, etc.
  - It's slow, inefficient and impossible for the driver to optimise in any sane fashion
- Fixed function pipeline (if possible)
  - FF is already being emulated by complex shaders; replace them with simpler ones!
  - You know what you need, the API doesn't

Start using...

- VBOs, PBOs (FBOs if available)
  - Store your geometry, texture and pixel data on the card and hint your intended usage to the API
- Shaders (where available)
  - Again, this usually leads to better performance on newer cards
- Plenty of reference documentation and tutorials available online

Vee-bee-what?

- Vertex buffer objects (VBO)
  - Store geometry data in memory allocated and managed by the driver
  - Significant performance boost for both static and dynamic geometry
- Pixel buffer objects (PBO)
  - Same thing, but for pixel data
- Framebuffer objects (FBO)
  - Render to texture, multiple render targets, modern framebuffer formats, etc.

Shaders?

- Vertex shader
  - Conceptually operates on single vertices
  - Stream processing model; no communication between vertices
  - Replaces lighting and application of fixed function matrices
  - Does not replace clipping
  - Outputs clip space vertices, usually from object space vertex input

Not just for vertices

- Fragment shader
  - Conceptually operates on single fragments
  - Stream processing model; no communication between fragments
  - Replaces color sum (texturing) and fog
  - Does not replace depth, alpha or stencil tests, alpha blending
  - Aware of its position on screen but cannot move (but newer cards can affect depth)

Shading languages

- ARB and vendor assembly variants
  - Just don't
- OpenGL Shading Language (GLSL)
  - Consistent, flexible, standardised
- Cg
  - Compiles to whatever shading language is available (except ATI-specific assembly)
  - Runtime freely available but closed source

Ideally GPU-friendly data

- Static; don't change geometry or texture data once uploaded
- Uniform; don't change render states, switch shaders or buffers
- Batched; draw everything in one go
- Efficient; use simple, fast shaders with few dependencies
- Don't draw stuff that won't be visible

More GPU-friendliness

- This above advice isn't realistic, but rather an ideal one should strive for
  - Your GPU will reward you for it
- It's assymetric, distributed computing
  - The less the CPU and GPU need to coordinate, the faster things become
  - The CPU is becoming the main bottleneck
  - Batch, batch, shader, buffer, batch

Resources

- The OpenGL SDK
  - http://www.opengl.org/sdk/
- Lighthouse 3D tutorials
  - http://www.lighthouse3d.com/opengl/
- NeHe Productions
  - http://nehe.gamedev.net/

Part 2

What is a rendering engine?

What is an engine?

- Programmer's view
  - A complex set of software modules, painstakingly created to enable accurate real-time rendering of large sets of 3D-data on our target platform(s)
- Artist's view
  - A black box created by incomprehensible, colour blind geeks, that somehow always renders my artwork incorrectly

The problem

You have this:

```
if (GLEW_ARB_vertex_buffer_object) {
  glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);
  glBindBufferARB(GL_ARRAY_BUFFER_ARB, bufferID);
  mapping = glMapBufferARB(GL_ARRAY_BUFFER_ARB,
                           GL_READ_WRITE_ARB);
  ...
```

...but you want something like this:

```
  scene.render(camera);
```

The solution

- Write lots of code
  - Wrap, automate, abstract, virtualise, etc.
- Raise the level of abstraction and ease of use at the cost of flexibility
- Know what you need
  - There is no such thing as a generic engine
- Modularity and OO are your friends
  - ...if you apply them correctly

Loss of flexibility

- Engine writing is about increasing ease of use at the cost of flexibility
  - Otherwise its API would be just as complex as the underlying one
  - Different applications need different parts of the total set of GPU and API features
  - This is why there aren't any generic rendering engines
  - All this is fairly obvious (but the myth of the generic rendering engine persists)

Finite variability

- Some abstractions are now universal
  - ...but won't be forever; things currently evolve very quickly in real-time CG
- Much time can be saved from following established practices
  - They exist for good reasons
  - But eternal glory to those who find fundamentally new techniques

What does an engine do?

- Manages the visual data set
  - Geometry, animations, shaders, textures
- Provides high-level graphics primitives
  - Multi-material meshes instead of triangles
- Hides the details of the graphics API
  - ...except when you need to get at them
- Most importantly, renders the data set
  - Decides where and how to render what

Common concepts

- Material
  - The entire set of data controlling the appearance of a geometry batch
- Technique
  - For systems supporting different levels of hardware, the material data for a given hardware profile
- Pass
  - The data set required to describe a single render pass of a material or technique

Common concepts

- Mesh
  - A lump of geometry, usually triangles, with one or many materials, together forming a discreet object (such as a table)
  - Usually the granularity level at which geometry is managed in an engine
- Sub Mesh
  - A subset of a mesh using a single material
- Mesh Instance
  - A mesh associated with a transformation

Common concepts, part 2

- Camera
  - Describing the current view frustum and transformations from worldspace into eye space (loosely analogous to position and lens shape)
- Viewport
  - The part of the screen you're rendering to
- Render Target
  - Something to which you can render; the screen, a texture, an offscreen buffer, etc.

Common concepts, part 3

- Render Operation
  - A single render batch, i.e. geometry buffer range, transformation and material
- Render Queue
  - Container for render operations
  - Collection separated from rendering allows sorting and other useful actions

Common concepts

- Scene
  - The top level spatial container of the visual data set (and in many applications, contains lots of other things as well)
- Scene Node
  - A node in the transformation hierarchy within a scene
  - Contains geometry, lights, particle systems and / or child nodes containing such things

Spatial partitioning

- Answers the question “what is here?”
  - ...without touching the entire scene
- This is useful when...
  - ...figuring out what to render from a given viewpoint or for a given effect
  - ...figuring out what lights affect the geometry you're rendering
  - ...doing lots of other things, some completely unrelated to rendering

Partitioning schemes

- Lots of different schemes exist
  - Octrees, BSP trees, quadtrees, AABB trees, portals, occluders, etc. ad infinitum, all with variants and combinations
- Two of these will be discussed here
- Different schemes perform well for different kinds of environments
  - Outdoor, indoor, at 10,000 ft, in space, 2D, 3D, 2.5D, large areas, tiny areas, etc.

Quadtrees

- Suitable for ground-based and other 2.5D environments
- Scene consists of subdivided 2D grid
- Automatically generated, no artist intervention required
- Fast and easy to code, but does not yield very precies results
  - Unless used with insane resolution

Example: Quadtrees

Image removed for copyright reasons

Portals

- Commonly used for indoor rendering
  - ...but has been effectively used for outdoor rendering as well
- Scene consists of cells and portals
- Other cells can only be seen through portals associated with them
- Efficient culling but a lot of clipping
- Burden of portal creation lies on artists

Example: Portals

Image removed for copyright reasons

“Real” scene graphs

- Tries to achieve everything through the graph hierarchy, complex node types, relations and traversal algorithms
- Please read Tom Forsyth's rant on the subject before attempting to implement one of these
- This isn't the kind of scene graph discussed in this lecture

“Basic” scene graphs

- Usually a simple tree structure
- Describes the relationships between discrete objects in the world
  - This wheel is attached here on this car
- Provides modelling of joints
  - Each node's position and orientation is relative to that of its parent
  - Your arm is a good example

Data gathering

- Spatial partitioning system query
  - Collect the intersection of scene and viewing frustum as efficiently as possible
- Generation of dynamic geometry
  - Billboards, particles, special effects
- Collection and sorting of operations
  - ...by material, by distance, by geometry buffer, by light source, by whether or not it's opaque, by something else or nothing

More data gathering

- Collection of relevant lights
  - Including per-light geometry for shadow rendering in complex environments
- Setup for shadow rendering
  - Stencil shadows need separate render passes (and sometimes new geometry)
  - Shadow maps need separate viewing frusta, queries and render targets
- Setup for reflection and refraction
  - Needs separate render passes or targets

Render ordering

- Don't change any state unless you absolutely have to
  - Especially shader, texture and geometry
  - If using fixed pipeline, many state changes now force hidden shader recompiles
- Therefore, order your data set to make the GPU happy
  - This is where render queues come in

More render ordering

- Render opaque geometry in front to back
  - Take advantage of z-culling; fragments that fail depth test are discarded unless they're using a depth-modifying shader
- Render transparent back to front
  - Not necessary for commutative blending
  - Even when operations aren't commutative, many applications get away with cheating

The resource manager

- Keeps track of geometry, materials, shaders, textures, animations, etc.
- Knows how to load and save data
- Does caching and delayed loading
  - Essential on consoles, good on PCs
- Lets different modules share data items without unneccessary duplication
  - You don't want to track this manually

Time for a break

Feel free to ask questions

Part 3

A simple rendering engine

Design overview

Meet Wendy

"What's your name?" he asked.
"Wendy Moira Angela Darling," she replied with some satisfaction.

Image removed for copyright reasons

The Wendy engine

- Wendy
  - OpenGL wrapper, scene graph, renderer core, rendering modules, user interface, demo system, etc.
- Moira
  - OS abstraction, math, binary and XML I/O, geometry and image processing, resource management, animation, signals, etc.
- Angela
  - Data conversion, viewing and editing tool

Design goals

- Modular and layered
  - Realised as a large number of modules rather than an integrated whole
- Extensible where appropriate
  - Extensible design takes time; use only where neccessary
- Quick and easy to use
  - Consistent design, even at the cost of (some) performance
  - Lots of error and warning messages

Core rendering classes

This part was not represented in slides

Material classes

This part was not represented in slides

Geometry classes

This part was not represented in slides

Static geometry

This part was not represented in slides

Scene graph classes

This part was not represented in slides

Resources: Theory

- Read books
  - Watt, Foley & van Dam, etc.
- Read whitepapers
  - SIGGRAPH, GDC, etc.
- Learn about your API
  - Features, shading languages, extensions, future directions, tricks, tweaks, etc.

Resources: Practice

- Read game development articles
  - Gamasutra, GameDev.net, etc.
- Look at existing engines
  - Ogre 3D, Irrlicht, CS, Q3, etc.
  - Most closed source engines have SDKs
- Write your own!
  - The best way to learn is by doing it

Questions

The End

Thank you for your time

Now go make something cool!

This material is available at:
http://www.elmindreda.org/opengl/