VU Entwurf und Programmierung einer Rendering-Engine

Introduction
What is a rendering engine?

- Things that make pixels?
- Game engine
  - Dynamic & static content
  - Tooling & content creation
  - Complex light & material system
  - Physics simulation
- Renderman
  - Light & material description
  - Artistic input
- Browser can also be seen as a rendering engine
  - Turns HTML into pixels
  - Includes real time requirements such as responsive layouting
Examples rendering engines

- Frostbite
- Effects, Level of Detail, Performance, Culling, Tooling, ....
Examples rendering engines

- Unreal
- Level editor, strong tooling, runs on every console & hardware
Game engines

- Have physically based lighting & materials and strive for realism
- But also have artistic input and support movie-like design
- Algorithms often fake approximations of “realism” in order to achieve Real-time performance

- Make it fast & look good
- Challenge: high quality visuals with limited dynamism & real time
Movie engines

- Physically based scene description and light
  - Advanced effects like dispersion, refraction ..
- Hacks and extensions to allow “unrealistic” artistic effects

- Make it exact & accurate
- Challenge: number of pixels/samples
Browser

- CAD tool on steroids
- Rendering + layouting engine
- Maximum dynamism
- Execution engine with user input

- Make it work quickly & robustly
- Challenge: everything is dynamic
Examples rendering engines

- Chrome
- Asynchronous loading, CSS/HTML parsing, error recovery
Parts of a rendering engine

- **Scene description language/system**
  - Developers can describe scenes in this language
  - Scene graph, language ruleset

- **Runtime system**
  - Interprets or compiles input language
  - Hardware dependent optimization
  - Translates scene description into renderer operations
  - Manages, allocates resources

- **Renderer**
  - Creates image from renderer operations
  - Raytracer, rasterizer

- API/Language design...
- Compiler tasks, OS Tasks,...
- effects, quality etc.
Limited dynamism

Movies
- mostly preprocessed
- no user input
- only time dependent

Game engine:
- Static parts (level, level lighting)
- Dynamic parts (AI, user input)

Interactive tools
- CAD/Browser/Editor
- Everything can change
- User can edit everything

Degree of dynamism:

Often little real-time interaction
- Dynamic stuff baked in
  - e.g. prebaked animation

Limited dynamism

Strong focus on dynamism

Everything is dynamic
Worst case: Consider current browser technology

- HTML tree very expressive
- Javascript can be used to rewrite large parts of the DOM (document object model)

[Demo: interactive changes in browsers, drawing demo]
Dynamism is important and complex

- Everything may change. The entire scene might be swapped out.
  - Consider page refresh in browsers or dynamic javascript content (e.g. D3)
- Resources must be allocated/deallocated cleanly

- ... while maintaining interactive performance
Dependencies: Incremental updates to the rescue

- Something changed -> refresh everything -> bad performance
- Smart Dependencies?
  - Every value becomes observable/change tracking system
  - Simple dependencies give rise to dependency graph
- Dependent change -> everything that depends on it
  - Incremental update
- Graph itself is dynamic
  - Structural change -> hard to handle, non-local effects
Abstraction

- Vulkan cube.cpp is 2900 lines of code
  - [https://github.com/googlesamples/vulkan-basic-samples/blob/master/demos/cube.cpp](https://github.com/googlesamples/vulkan-basic-samples/blob/master/demos/cube.cpp)
- Clearly, there is a need for abstraction
- Various types
  - macros
  - Reusable utility functions (e.g. createShader)
  - Notation for objects? Object list
  - Common abstraction: scene graph
    - e.g. HTML, Markdown, UI elements, scene entity tree
What is a scene graph

- Most general scene description
- At first glance: feel natural
- But also supports the level of dynamism & abstraction required
  - Not that easy, but we will see how....
Naive scene graph implementation

- Implementation -> traverse, allocate resources on the fly
  - OpenGL intermediate mode view
    - Command-buffer implementation possible.

- What happens if scene is dynamic: simple -> retraverse (cache resource allocations)
  - Question: when to free resources: garbage collection
  - Question: when to retraverse?
  - Answer: don’t know -> always
Better scene graph implementation

- Finding appropriate abstraction is challenging.
- Abstraction in combination with dynamism is even harder.
- How to provide expressive/easy to use APIs?

Try to combine those with optimal performance / best hardware utilization!
Scene graph rendering (evaluating the graph)

Translating scene graph input structure directly (in one step) into graphics instructions is hard. Can we approach the problem differently?

Comparison: it is hard to translate c++ directly into machine code. Most compilers use appropriate intermediate representation

- Is there a common intermediate representation in our domain?
- We focus on rasterizer-specific features
  - Materials/BRDFs in ray tracer VS cullmode stencil in GL

```python
list { RenderObject1,
      RenderObject2,
      ...
    }
```

```
for ro in renderObjects:
    Graphics.setViewTrafo ro.Trafo
    Graphics.setShader ro.Shader
    Graphics.render ro.Geometry
```
Requirements for Rendering Engines

- Easy to use and extend
- Translation of scene description into graphics commands

Performance
- Utilize graphics hardware as best as possible!!!
- Responsiveness
- High-frequency changes
Graphics API Insights required

- In OpenGL there are dozens of ways to solve a problem inefficiently.

```c
index = (index + 1) % 2;
nextIndex = (index + 1) % 2;

glReadBuffer(GL_FRONT);

glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, pboIds[index]);
glReadPixels(0, 0, WIDTH, HEIGHT, GL_BGRA, GL_UNSIGNED_BYTE, 0);
glBindBufferARB(GL PIXEL PACK BUFFER_ARB, pboIds[nextIndex]);
GLubyte* ptr = (GLubyte*)glMapBufferARB(GL_PIXEL_PACK_BUFFER_ARB,
                                        GL_READ_ONLY_ARB);
if(ptr)
{
    processPixels(ptr, ...);
    glUnmapBufferARB(GL_PIXEL_PACK_BUFFER_ARB);
}

glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, 0);
```

http://www.songho.ca/opengl/gl_pbo.html
Graphics API Insights required

- There are many approaches for uploading data to OpenGL. Even on single hardware there are significant differences...
- Dozens talks, forum discussions, e.g.:

Low level optimizations

- For high performance, we need to know the cost of abstraction:
- Examples:
  - Loop overhead in image processing: memcpy vs copying pixel by pixel
  - Allocation overhead: e.g. accidental allocations in scene graph traversal
  - Virtual calls in performance critical code?
  - How is multiple inheritance implemented? What are the costs

Our OpenGL renderer has a custom AMD64 assembler...

```plaintext
member x.Mov(target : Register, value : uint32) =
    if target < Register.XMM0 then
        let tb = target |> byte
        if tb >= 8uy then
            let tb = tb - 8uy
            let rex = 0x41uy
            writer.Write(rex)
            writer.Write(0xB8uy + tb)
        else
            writer.Write(0xB8uy + tb)
    else
        writer.Write value

else
    x.Mov(Register.Rax, value)
    x.Mov(target, Register.Rax, false)
```
Towards interactive lighting systems

Armed with technical tools for

- Abstraction
- Mechanisms for handling dynamism
- Algorithms and datastructures
- Low level understanding and techniques

We turn shift our focus towards tooling for real-world interactive lighting simulation....
Later in this lecture we return to more traditional rendering techniques again...

Two lectures by lui the king. Topics:
- Material systems
- Instant radiosity
- Deferred Rendering
- Physically based shading
Topics of this LV

● Render scenes efficiently using graphics hardware
  ○ Graphics Hardware and API (recap)
  ○ Common infrastructure for rasterizer-based rendering backend (intermediate language)
  ○ Scene description
  ○ Optimization techniques

● Rendering scenes nicely
  ○ Using lighting and material systems
  ○ Concrete techniques such as shadow mapping is part of other lectures
  ○ Here we focus on practical implementations thereof
Focus of this LV

- Tools/Algorithms/Concepts to implement rendering engines
- Hardware/Graphics API insights
- Performance considerations
  - optimizations (how to pack buffers etc)
  - Costs of programming language abstractions (e.g. can we afford virtual function calls?)
- Approaches for implementing lighting/material systems

Not content of this LV:
- Graphics API Tutorial
- How to use existing engines
- How to implement tooling (e.g. level editor, material editor)
<table>
<thead>
<tr>
<th>Engine tools, e.g. Level editor</th>
<th>Game titles</th>
</tr>
</thead>
</table>

**Game Engine**

- File loaders, Scene managers....

**Rendering Engine internals (this LV)**

1. Materials, Lighting (e.g. Lightmap generation)
2. Utilities such as culling, transparency sorting, texture packing,....
3. Notation of renderable things
4. Graphics API
5. GPU

**The big picture.....**
### Rendering Engine internals

<table>
<thead>
<tr>
<th>Materials, Lighting (e.g. lightmap generation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilities such as culling, transparency sorting, texture packing,....</td>
</tr>
<tr>
<td>Notation of renderable things</td>
</tr>
<tr>
<td>Graphics API</td>
</tr>
<tr>
<td>GPU</td>
</tr>
</tbody>
</table>

High-level Abstraction:
Graphics scenes, Shaders...

**efficient mapping?**

High-level Abstraction:
Graphics scenes, Shaders...
Bridging the gap

Two approaches:

- Condense scene to minimal representation and then map it to hardware
- Given hardware, what utilities can we expose to build more powerful tools?
Bridging the gap

Two approaches:

- Condense scene to minimal representation and then map it hardware
- Given hardware, what utilities can we expose to build more powerful tools?
Upcoming topics I

● Top down: Traditional scene graph systems
  ○ Implementation techniques
  ○ Advantages and disadvantages

● Bottom up: Hardware capabilities and their implications

● Bridging the gap: Towards a rendering engine runtime system
  ○ How to model scene data in order to map it to graphics hardware efficiently.

● Low-level optimizations for efficient graphics programming
  ○ Towards adaptive optimizations

● Algorithmic optimizations
  ○ Algorithms and data structures for rendering engines
Upcoming topics II

- Practical topics for rendering engines
  - Mesh representation
  - Performance considerations
  - Precision considerations

- Domain specific languages for
  - Dynamic data
  - Scene representation (in presence of dynamic data)
  - Shader programming

- Towards real-time high quality lighting
  - Global Illumination, Material models
  - Physically based shading
  - Deferred Shading
  - Instant radiosity, Texture packing
Videos

- Siggraph 2016: Surface-only liquids
  - offline-rendering
  - https://www.youtube.com/watch?v=9gUSmYRl8B8
- Battlefield 1 gameplay
  - User input
  - https://www.youtube.com/watch?v=-NxAzWAM9Hc
- Unity game dev speed-up
  - Scene description & scripting languages
  - https://www.youtube.com/watch?v=fiHRxD1yE4Y