Game Engines: Why and What?
Dan White
Technical Director
Pipeworks
danw@pipeworks.com
Message

- As you learn techniques, consider how they can be integrated into a production pipeline.
Sense of scale…

- Video games word-wide: $110 billion business in 2015
  - Main driver of 3D graphics hardware
    - Not simulations or NORAD
  - Main driver of 3D rendering software
    - Not Hollywood
Budgets and Timeline

- Individual Game budgets are big.
  - Current AAA titles:
    - $30 million budget, 2+ year dev cycle.
    - GTAV = 1 $Billion, $265 million to develop.
  - Even mobile games: $500 thousand
  - Project completion date is very important.
    - Particularly for licensed properties, sports, holiday launches, or anything with advertising.
  - Schedules from 3 month to 5+ years
Managerial Ideal

Money → Game development organization → Product

Product → More Money
Olde Tyme Game Making

35 years ago, a game might have 1 programmer, and maybe 1 artist

They were written...like a novel is written.
A current mid/large size project looks very different…

On large projects, programmers are a small part
On smaller projects, a larger part
This doesn’t even include QA
A different approach

Jumpman
Atari 800
circa 1983:

This game had an EDITOR!
At the time, this seemed revolutionary to me.
Doom Engine cemented the idea

Doom released in 1993

- Spawned HeXen, Heretic and so on.
- Started the idea that an engine is potentially valuable on its own.
Game Engine Concept

- A collection of reusable tools and runtime for making games.
- An **integration** platform for all the different features of the game.
- Used by the different disciplines to do their work.
- Ideal is you don't need engineers to make a game.
  - After all, you don't need engineers to write a book or design a building
  - Still plenty of work for engineers!!
How the team works…

- Artists
- Animators
- Designers
- Scripters
- Audio Designers
- Gameplay Programmers
- Engine
- Product
- Game specific code
Production should be a pipeline

Tool chain  Runtime  Back End

Import Assets  Distribute
Edit Game Data  Load
Bundle for Runtime  Simulate

Collect Data  Analyze

Present
Who are the engine users?

- **Artists**
  - People with art talent and a wide range of technical skills.
  - They want to make assets in a DCC tool (e.g. Maya) and put them in the game.
  - Need to be able to see the final result.

- **Designers**
  - Also a wide range of technical skills.
  - They want to arrange content to produce fun.
    - Place objects, manipulate values, script actions.

- **Game programmers**
  - Engineers writing code specific to the game you are making.
  - Organizationally separate from the people making the engine.
Everybody wants…

- **Easy of use:**
  - Intuitive UI
  - Stability

- **Fast iteration times:**
  - Make a change
  - See the result
  - Make another change

- **Iteration is the KEY to making games good!**
  - Good: Play in editor…10 second loop
  - Bad: Bundle and launch full game…8 minute loop
Key Engine Features

- Rendering
- Serialization
  - Asset loading
- Object simulation
- Camera & Controls
- UI system
  - Need for shell and HUD
    - Do not underestimate!
Other Features

- Asset Management
  - Versioning
- Build system
- Sound
- Animation
- Physics
  - Cloth
- (Simple) Networking
- And on and on...
  - Feature set grows over time
  - No feature is minor when your game requires it!
Spigot
It’s a hard problem

- Engines take many years to develop.
  - Many fail.
- **Not** because of the difficulty of basic research.
  - Most features begin in academia, or with graphics card manufactures.
- Hard part is integration into a usable system.
Key Integration Points For a Feature

• Tool Chain
  ◦ How do you get external assets into the game?
  ◦ Example Meshes:
    • Built in Maya
    • Exported as FBX
    • Imported into editor for display

• In-editor UI
  ◦ How will artists and designers configure your feature?
    • Visual editing is better.
    • This is usually the most time consuming part.
  ◦ How do people preview and iterate?
More Integration Points

- **Storage & Loading**
  - How are your configuration parameters stored by the editor?
    - Ideally, this will be text so you can diff results.
  - How will the data be preprocessed at build time?
    - Runtime data is highly optimized, and this can take a long time.
  - How will data be quickly loaded at runtime?
    - May need to stream it.
More Integration Points

- **Runtime**
  - What code runs in the actual game?
  - Games are real-time systems, hence predictable performance is key.
    - Do not amortize costs in algorithms.
  - What resources do you need:
    - Memory
    - Texture Memory
    - CPU
    - GPU
  - Can you make the code parallel?
- **Interactions with other features.**
Interactions

- How does a feature interact with other features?
  - Sometimes features work against each other.
  - You can’t just change the rendering pipeline cause you want to.
- Usually this is the **hardest** (if not the most time consuming) part.
Example Feature: Particle Systems
Basic Idea

- Render a sprite many many times to produce smoke, explosions, etc.
- Can produce a wide variety of effects with small amounts of source art.
  - Great for engineers who want to make cool stuff.
- Generally not physically based.
  - Not widely studied in academia.
Tool Chain

- Particle system:
  - Input is textures.
    - Made in Photoshop
    - Imported as PSD, JPEG, etc.
Particle systems:
- Invent the concept of “emitters” which produce particles.
- Using existing UI for materials.
  - In Unreal, this is a graph-based system.
- Use a property sheet for parameters.
- Use a visual editor for curves.
  - Supports splines.
Unreal Material Editor
Unreal Cascade Particle Editor
Storage, Preprocessing, Loading

- Particle system:
  - Editor description could be text.
  - Runtime data is small.
    - Store as binary POD.
    - Reference materials, which pull in textures.
      - Converted to DXT or similar.
Runtime

- **Particles:**
  - Straightforward to write.
  - Render large numbers of dynamically updated quads or primitives.
    - It’s a bad idea to call DX/OpenGL many times.
      - Need to coalesce in vertex buffers or display lists.
  - Extremely fill intensive.
  - Updating many particles is CPU intensive.
    - Particle systems very suitable for parallel processing.
Interactions

- Particles are typically translucent.
  - Rendered with alpha.
  - Should they write to the Z-buffer? Probably not.
    - You will likely have to sort them relative to other objects.

- Depth complexity of particle systems is very high.
  - Will destroy your fill rate if they are close to the camera.
    - Have to LOD particles as you get close, or limit camera.
  - May implement lower-res render to texture.

- If you have depth based fog, do you apply fog to the particles?

- Deferred rendering
  - A screen space rendering technique that uses passes.
  - Doesn’t handle alpha. What to do?
    - Dithering
    - Separate forward rendering pass.
Some observations…

- The editor and tool chain are much more complex than the runtime.
  - Lots of UI work!
- The whole team works with the tool chain, but only engineers work with the runtime.
  - Productivity payoffs for improved tools can be very large.
Downside of Engines

- Cost.
- Produces external dependencies.
- Poor support for particular genres.
  - e.g. RTS
- Generic performance may not be as good as special code.

These are issues as old as software.
Hardware Evolution: Where are we going?

<table>
<thead>
<tr>
<th>CPU</th>
<th>NES</th>
<th>PS1</th>
<th>PS2</th>
<th>PS3</th>
<th>Current PC</th>
<th>Xbox One</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOS Tech 6502 1.79MHz</td>
<td>MIPS R3000A-32-bit RISC chip at 33 MHz</td>
<td>294 MHz MIPS &quot;Emotion Engine&quot;</td>
<td>3.2 GHz POWER PPE, seven 3.2 GHz SPEs</td>
<td>8 Cores at 3 GHz</td>
<td>8 Core AMD custom CPU Frequency: 1.75 GHz</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>66 MIPS vector math unit on CPU</td>
<td>147 MHz &quot;Graphics Synthesizer&quot;</td>
<td>550 MHz based on Nvidia G70</td>
<td>Gerforce 650 1058 Mhz 384 Cores</td>
<td>853 MHz GPU Custom AMD</td>
<td></td>
</tr>
<tr>
<td>Cartridge</td>
<td>CD</td>
<td>DVD</td>
<td>Blu-Ray</td>
<td>HDD</td>
<td>HDD</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- MOS: MOS Tech 6502
- CPU: CPU Frequency: 1.75 GHz
- GPU: "Graphics Synthesizer" based on Nvidia G70
Content is King

- We have already reached the sweet spot for most game features:
  - 4 enemies to 8 → big difference
  - 64 enemies to 128 → small difference
- PS2 was the tipping point between ability to display content and ability to make it.
- We are now firmly in the era where content creation cost is the driving factor.
- Runtime performance gets even less important.
Beyond the Engine: The Back End

- **Disruptive force since late 2000’s:**
  - Piracy, and the need to control it
  - Digital Distribution
  - Success of MMORPGs (e.g. WOW)
  - High broadband penetration
  - Microtransactions as business model
  - Resurgence of the PC, and emergence of mobile
  - Abundant web technology & infrastructure

- **Result has been the rise of games with a “back end” component**
  - Means: the game connects to a database
Implementation

- Backend is typically not part of the game engine.
- Uses technologies not traditionally part of game development.
- Primary reason: “mainstream” software development tools can be used.
  - Node.js
  - .NET
  - CodeIgniter
Back End Implications

• Once you connect to a database, many things become easy:
  ◦ Cloud based save load
  ◦ Multiplayer lobbies & leaderboards
    • But not synchronous MP
  ◦ Social integration
  ◦ Freemium economy & transactions
  ◦ Piracy protection
  ◦ Telemetry, which has changed game design forever
• Typically implemented via a HTTP/HTTPS & REST
  ◦ Most common back ends are PHP, but all kinds used
  ◦ Most common database is MySQL, but nosql is gaining
    • Games are much more write heavy than other Web apps
  ◦ Scalability is a problem
• 3rd party hosting services (AWS, Rackspace) used a lot
• One of the biggest areas of active “research.”
What it takes…

- Games are a serious career for people who are serious about it.
- Game programming involves skills missing from a traditional CS program:
  - Mathematical modeling, vector math
  - Simulation & physics
  - Graphics, particularly special effects
    - This is why I’m so excited about this class!
- Programming tasks often end up with a complex integration step.
  - Strong programming skills are essential.
Message Recap

- As you learn techniques, consider how they can be integrated into a production pipeline.
- Iteration! Iteration! Iteration!