Evolution of a Game Engine

Eric Lengyel

TERATHON SOFTWARE

Outline

- Introduction to game engines
- Creating a game engine
  - Software engineering practices
  - Major parts of an engine
- Maintaining a game engine over time
  - Example case: C4 Engine graphics and audio
- Preparing for the future
  - Multithreading

What is a Game Engine?

- The word “engine” is used for a lot of things
  - Generally, any significant piece of self-contained code that does something useful
- The term “graphics engine” often used to describe software that provides high-level rendering capabilities
  - Provides more functionality than plain OpenGL or DirectX
  - Could support things like a scene graph, shadows, visibility determination, etc.
  - Can be a very complicated piece of software

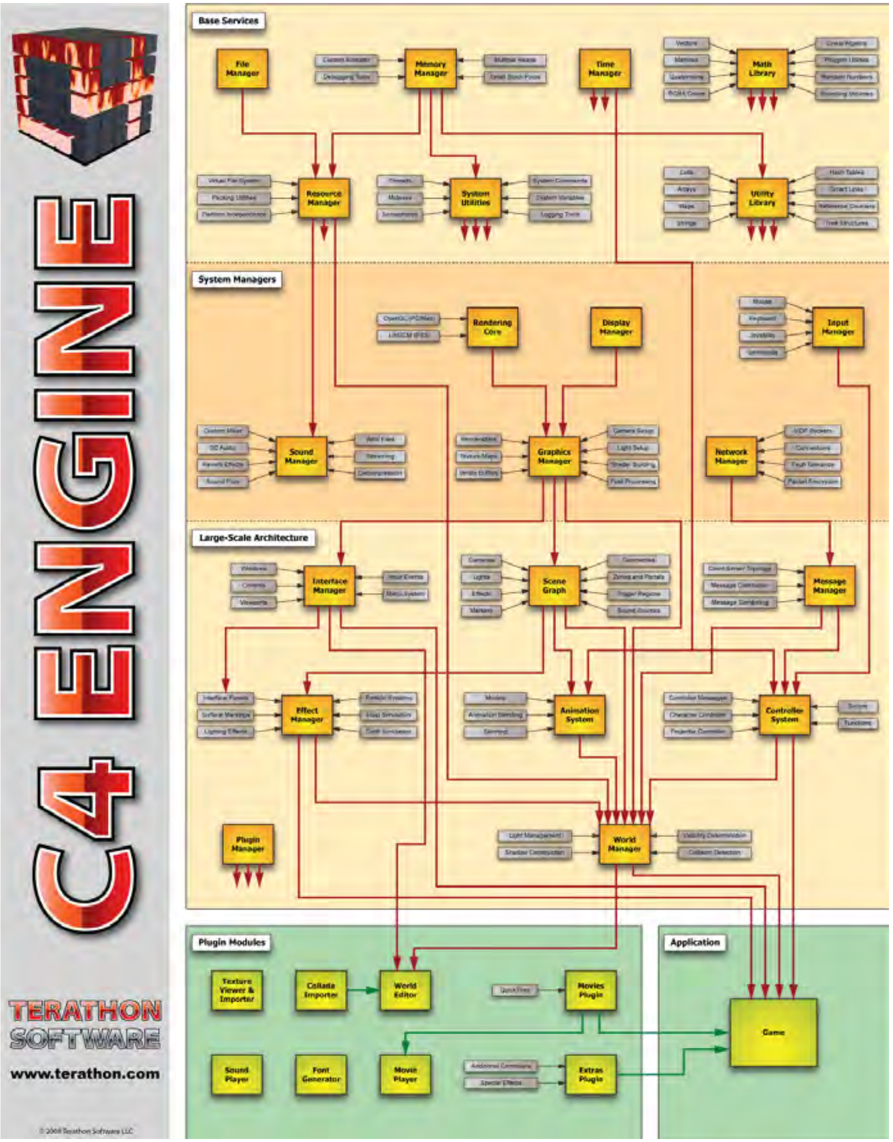
What is a Game Engine?

- The term “game engine” typically means a graphics engine plus a lot of other stuff that most games need
  - Audio, music, 3D sound effects
  - Input (keyboards, mice, other controllers)
  - Networking for multiplayer games
  - Resource management
- Game engines usually provide several higher-level features as well
  - Physics, animation, special effects, AI, etc.

Creating a Game Engine

- A game engine can be an extremely complex piece of software
- It’s important to employ intelligent software engineering practices
- Games typically have greater performance requirements and resource constraints compared to other large software projects
- Game programmers often want more control over the computer, and will reduce external dependencies to get it

C4 Engine Architecture



Layer 1
Base Services

Layer 2
System Managers

Layer 3
Large-Scale Architecture

Layer 4
Game and Tools

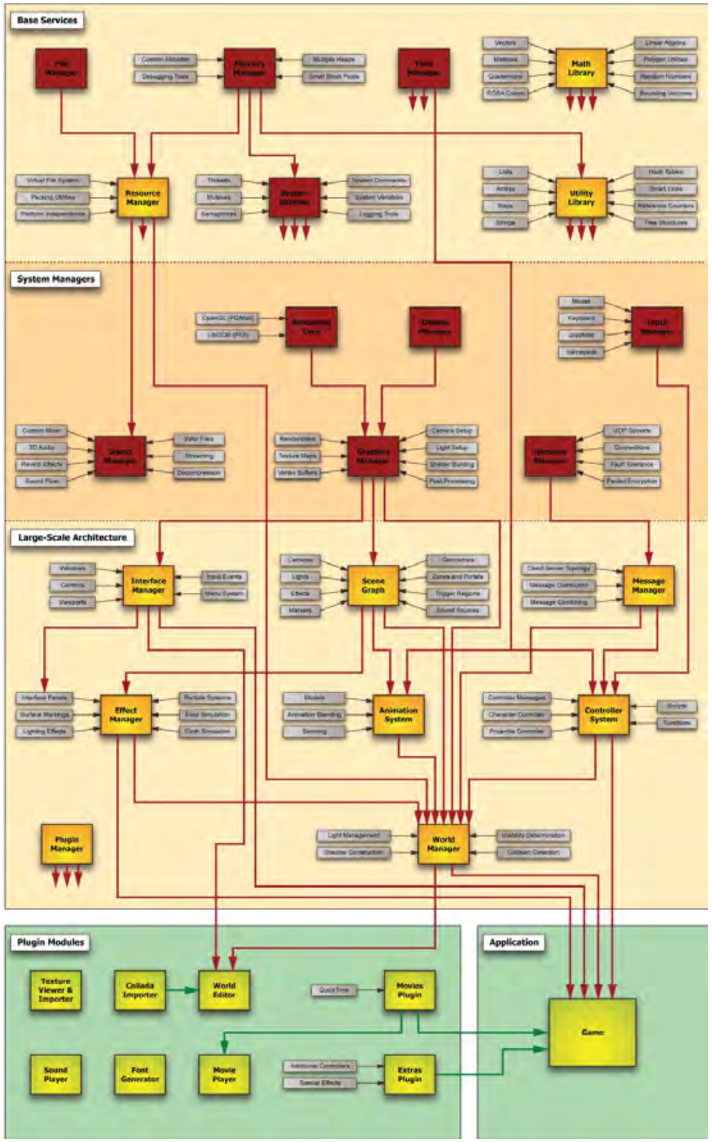
Software Engineering

- Component isolation
  - Easier to accomplish at lower levels
  - Should be possible to replace a component without changing other code
- Layered architecture
  - Code in one layer should only make calls to code in its own layer or lower layers
- Good class design
  - Use C++ inheritance and virtual functions
  - Don’t duplicate functionality

Software Engineering

- Design the engine code for cross-platform deployment
  - Even if the engine will only be deployed on a single platform
  - Isolate platform-dependent code as much as possible
    - Usually code that calls into the operating system
    - Can also pertain to code that uses external libraries
  - The engine should be aware of byte order
    - Loading resources from disk
    - Sending/receiving data over the network
    - Mac changed from big endian to little endian!

OS-Dependent Components



- Dark red components make calls into the OS

- Higher layers not aware of underlying OS

Demand on Computer Resources

- Games typically eat up all the resources that are available on a computer
- Performance consistency can be sensitive to environmental conditions beyond the control of the game engine
  - An engine can have different performance characteristics on different platforms
- Therefore, engine programmers tend to implement everything they can themselves
  - Remove dependencies on external code, even the standard libraries!

Low-level Engine Components

- Memory manager
  - Override standard new and delete operators
  - Some consoles do not have virtual memory
  - Must ensure exactly the same behavior on all platforms
- Containers library
  - Replaces STL
  - Implemented in such a way that memory allocation is strictly controlled, or for many containers, is eliminated

Low-level Engine Components

- Math Library
  - Vectors, matrices, quaternions, colors, etc.
- File Manager
  - Platform-independent access to disk
- Resource Manager
  - Generic system for tracking all types of data read from the disk
- Time Manager
  - Platform-independent timing facilities
- System Utilities
  - Threads, mutexes, signals, etc.

Building an Engine from Scratch

- First need a basic shell running
  - Most of the low-level components will need to be written very early in the development process
  - The engine needs a simple event loop providing per-platform responses to input events
  - Some simple form of output will be necessary until a more capable graphics system is available
- Larger components can then be developed
  - These make up the second layer of the engine
  - Low-level graphics, sound, networking, input

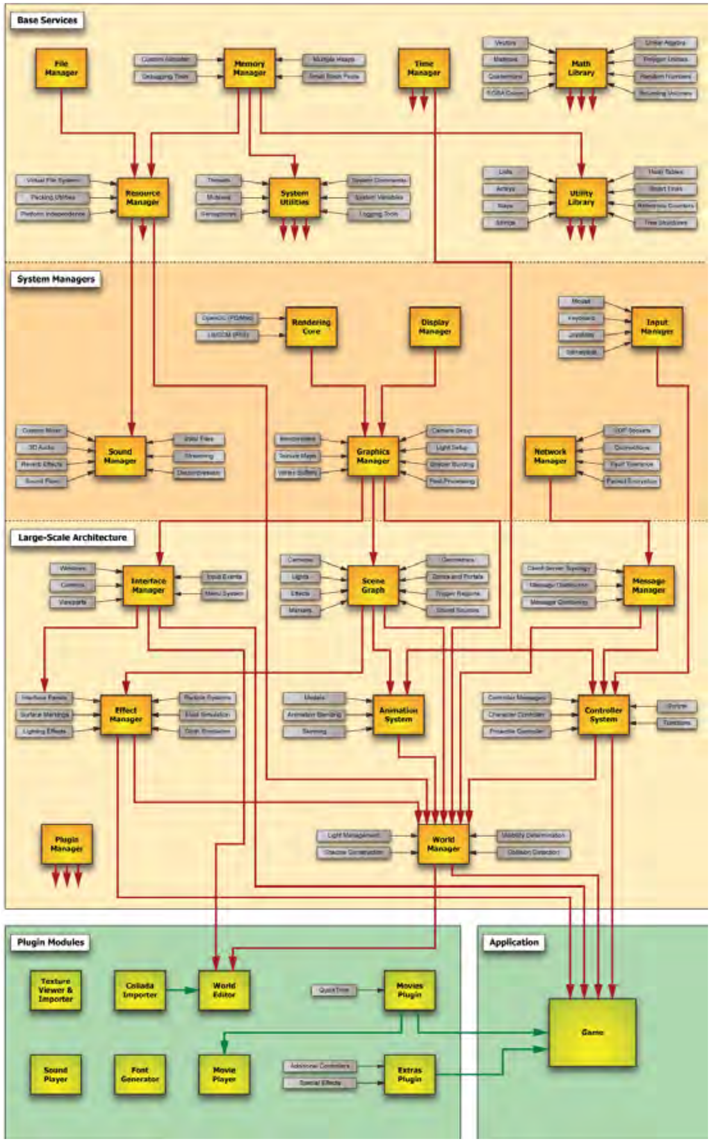
Component Refinement

- The first implementation of a component will usually be thrown out!
  - When you're implementing a system for the first time, new design considerations will often pop up that couldn't really be foreseen
  - Early in development, many components could be trashed and restarted several times
  - Later, components tend to be rewritten much less frequently, but it still happens
    - Perhaps new capabilities are required or new technologies become available

Component Refinement

- Frequency of rewrites decreases with layer height
  - In the C4 Engine, almost all 1st and 2nd layer components have been rewritten at least once
    - Good isolation means higher layers are not affected
  - Many high-level components have not been rewritten in their entirety
    - But many parts of these components have been refined at one time or another

Component Rewrites



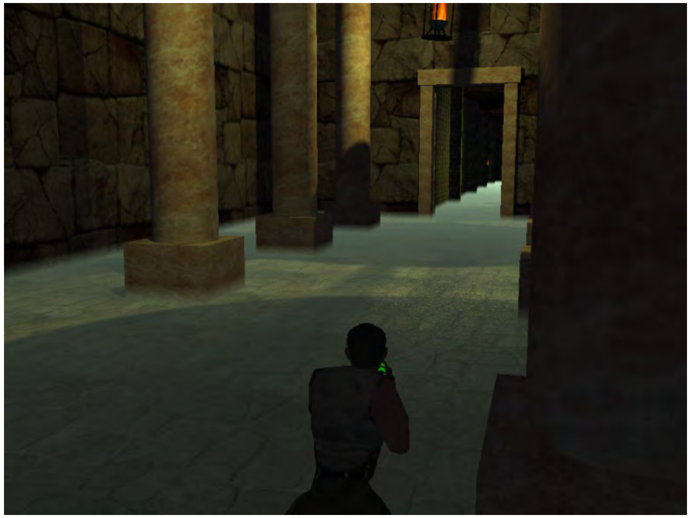
Memory Mgr:	1999, 2000, 2007
Resource Mgr:	2000, 2004, 2007
Graphics Mgr:	1999, 2000, 2001, 2005, 2007
Sound Mgr:	1999, 2001, 2007
Input Mgr:	2001, 2003, 2005
Network Mgr:	1999, 2001
Interface Mgr:	1999, 2004
Animation Sys:	2000, 2006

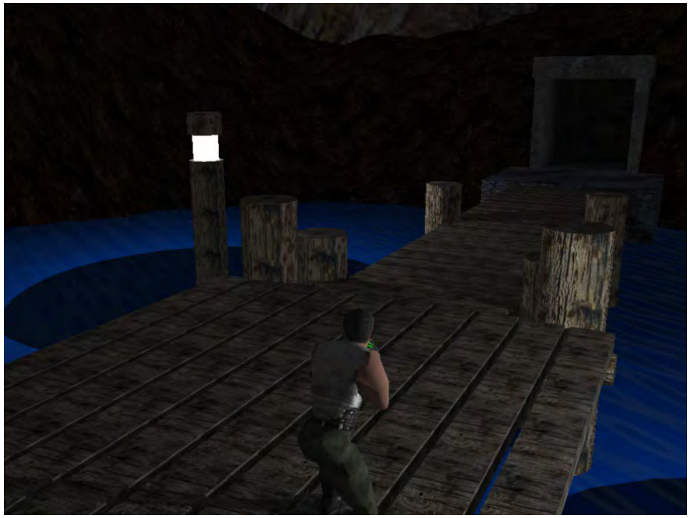
Evolution of Graphics Technology

- 1999
  - Baked light maps
  - Multi-pass rendering to achieve special shading
  - Limited stencil shadows
  - Limited dynamic lights

Evolution of Graphics Technology

- 2001
  - Fully dynamic lights
  - Robust stencil shadows on everything
  - Projected shadows
  - Separation of ambient lighting term
  - Normal mapping
  - Required GeForce 256+ or Radeon 8500+



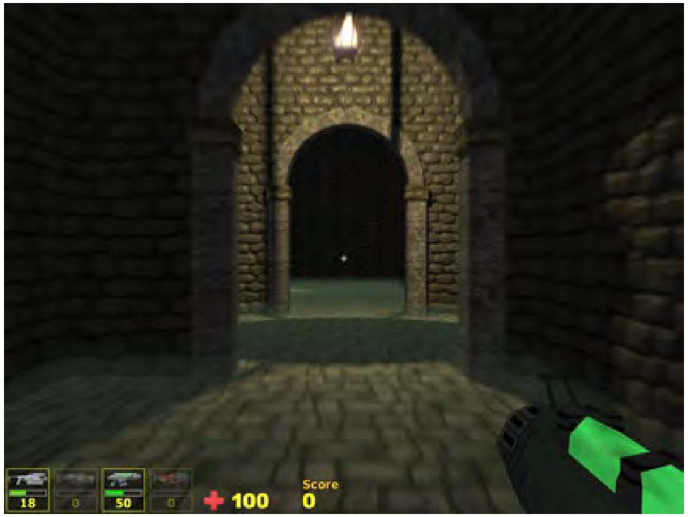


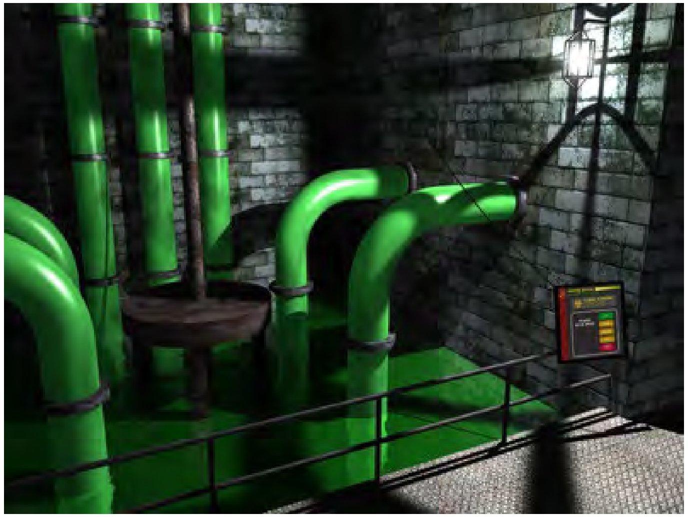
Evolution of Graphics Technology

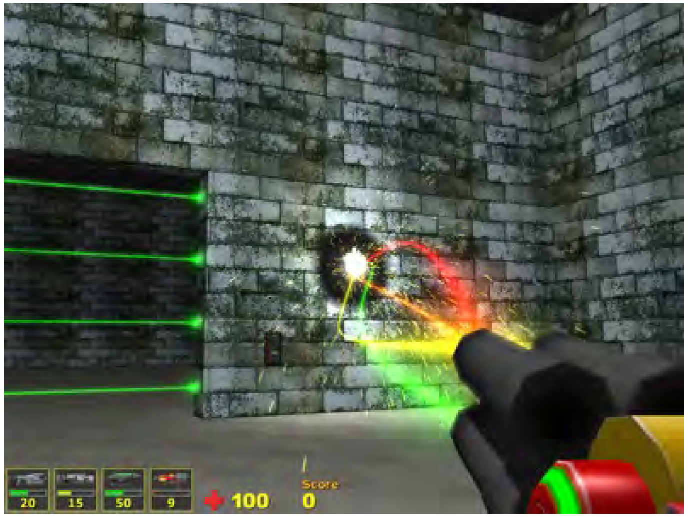
- 2005
  - Shadow maps
  - Reflection and refraction portals
  - Parallax mapping
  - Ambient light volumes
  - Required GeForce 3+ or Radeon 9500+

Evolution of Graphics Technology

- 2006–2007
  - Post-processing (motion blur, glow)
  - Lots of new special effects (fire, panels, fog, special particles)
  - GF 3-4 support dropped

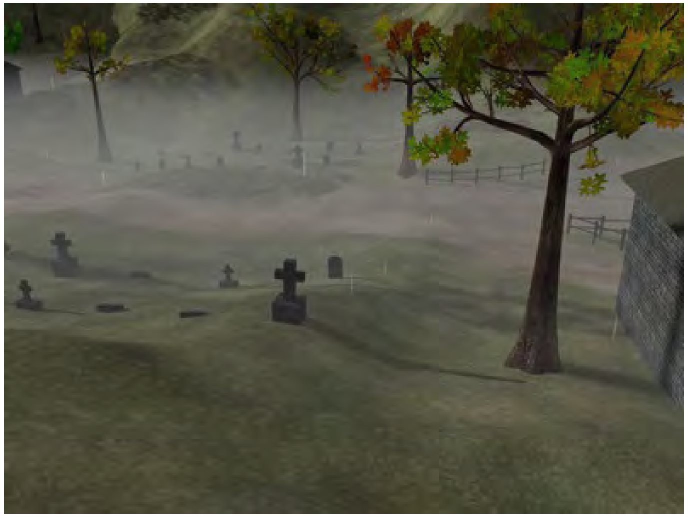


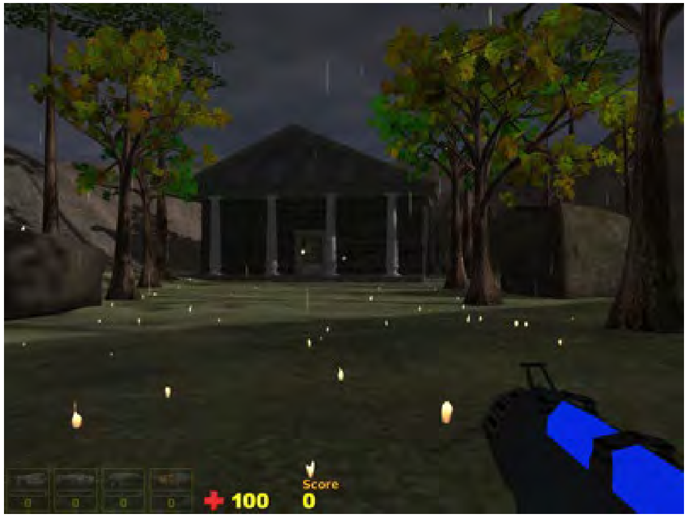


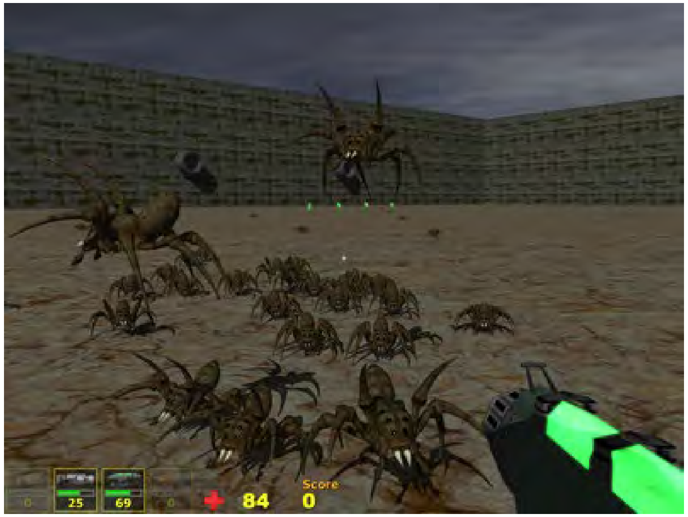


Evolution of Graphics Technology

- 2008
  - More advanced ambient shading
  - Large-scale terrain and outdoor scenes
  - Dense foliage
  - Built-in physics





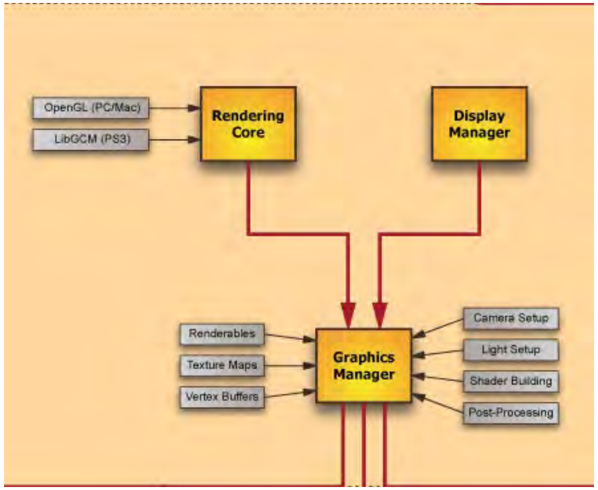


Cross-Platform Graphics

- OpenGL available on PC and Mac
  - Unfortunately, only Nvidia implements support for latest GPU features in OpenGL
  - OpenGL 3 is vaporware!
- OpenGL ES supported on PS3
  - But not used by any serious developers
  - Lower-level library used instead
- Excluding Nvidia, general trend is waning support for OpenGL games
  - Cross-platform solution going down the tubes

Cross-Platform Graphics

- Lack of cross-platform graphics required changes to C4 Engine
  - The code that accesses the graphics hardware has been separated from the Graphics Manager
  - Makes future DirectX version easier



Shadowing Techniques

- C4 has been using stencil shadows
  - Robust technique
  - Requires closed meshes
  - Produces silhouette faceting
  - Uses lots of hardware fill power
- C4 currently has limited shadow mapping
  - Mostly robust technique
  - Works for non-closed meshes
  - Much better for alpha-tested geometry
  - Produces artifacts (“shadow acne”)
  - Resolution dependency

Shadow Mapping

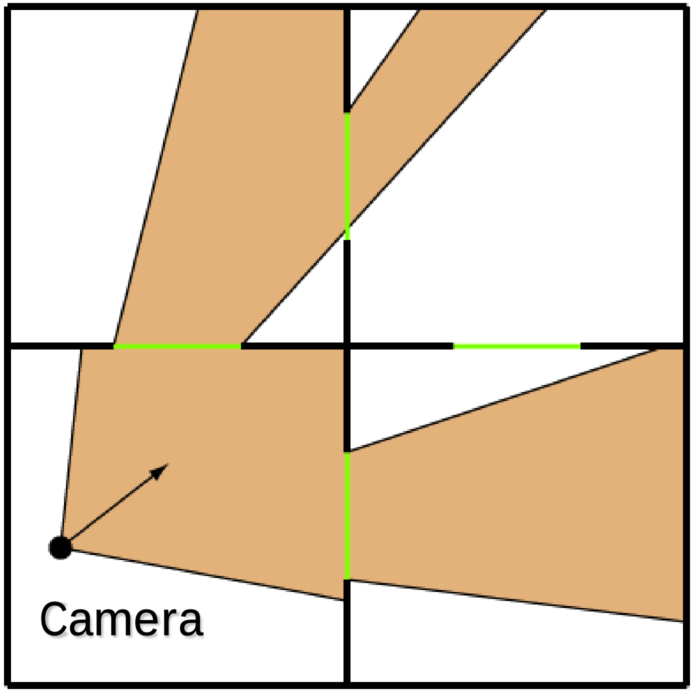
- C4 moving toward dynamic shadow maps
  - Orthogonal to stencil shadows
  - Hard to do for point lights
  - Will require more memory
  - Potential for higher performance
  - The only option for dense foliage

Higher-Level Graphics Systems

- Large-scale visibility determination
  - Essential for good performance
  - World must be organized into a structure that can be used to quickly figure out what is visible
    - Old-school BSP tree
    - Quad tree, octree
    - Hierarchical bounding volume tree (BVH)
    - Portal system
    - Combinations of the above

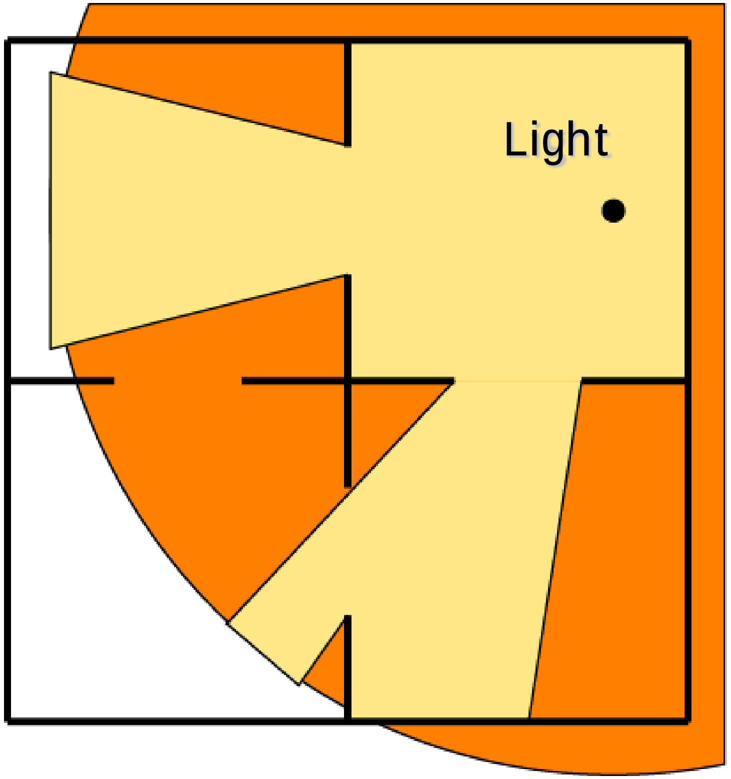
Portal System

- A powerful technique for any size world
  - World is divided into zones connected by portals
  - Only accesses parts of scene that are actually visible
  - Works well with a per-zone BVH
  - Easy to implement basic algorithm



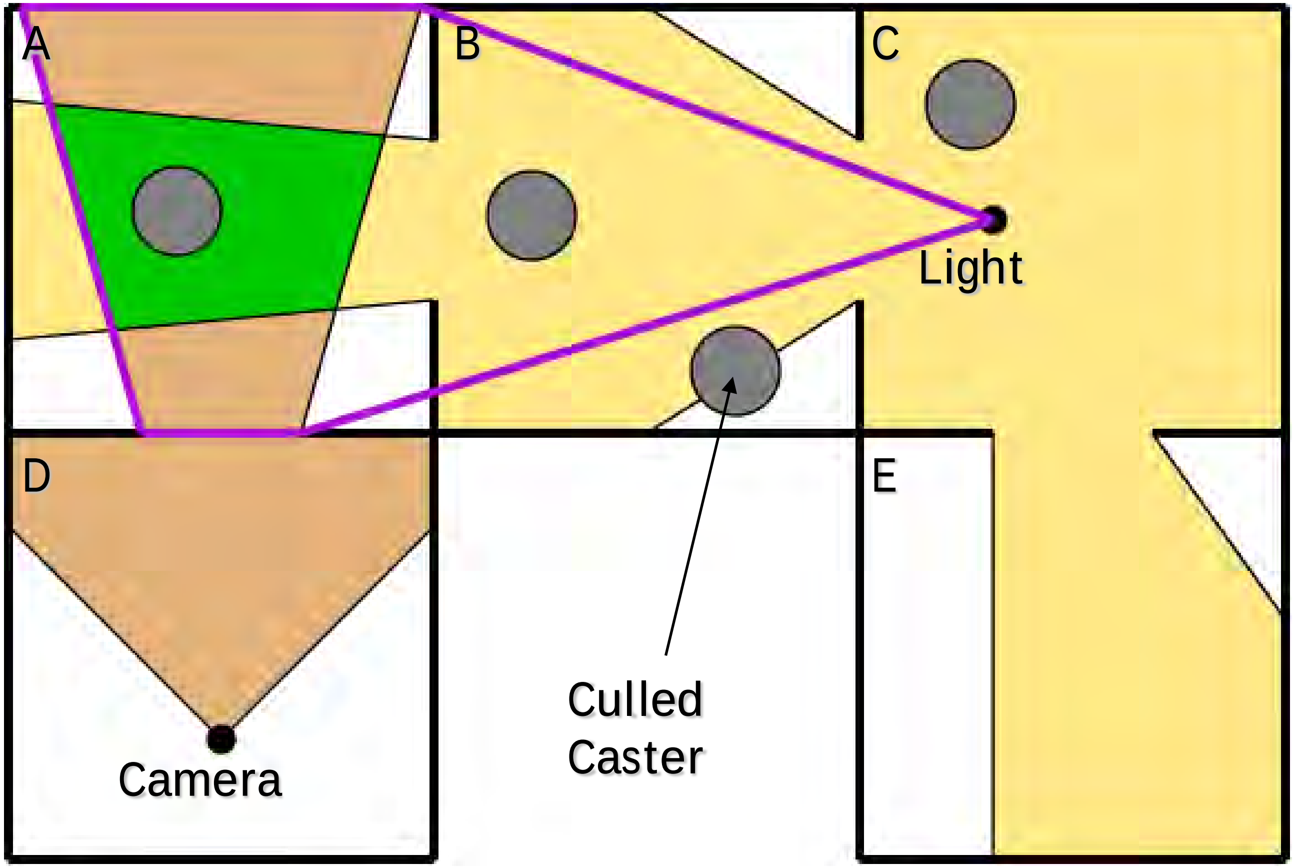
Portal System

- Also used for lighting
  - Engine can use portals to determine set of visible lights
  - All other lights in the world are ignored



Portal System

- Becomes very complicated when dynamic shadows are thrown in



Evolution of Audio Technology

- Basic audio support
  - Play mono and stereo sound data
  - Streaming from disk
  - Simulate 3D positioning
- Environmental Audio Effects (EAX)
  - Hardware acceleration
  - Reflection, reverb, absorption
  - Frequency-dependent effects
  - Unfortunately, not well-supported
  - Not cross-platform

Evolution of Audio Technology

- No audio API standard
  - OpenAL was a valiant attempt, but never really took off
  - DirectSound now replaced by XAudio
    - At least this covers Windows and Xbox 360
  - Only high-performance path on Mac is CoreAudio
  - PS3 has its own audio library

Evolution of Audio Technology

- C4 now uses custom mixer
  - Uses minimal amount of functionality in the various audio libraries
  - Provides consistent results on all platforms
  - Full-featured
    - Reflection, reverb
    - Atmospheric absorption
    - Doppler, frequency effects
  - Perfect fit for dual-core processors!
    - Audio processing decoupled from rendering loop
    - Streaming nature easy on memory caches

Multi-Core Processors

- The near-term future of computing
  - Dual-core is already common
  - Quad-core through 16-core common soon
  - PS3’s Cell processor has 8 special cores
  - GPUs becoming general-purpose processors with lots of cores
- Game engines need to be designed to take advantage of this power
  - Tasks need to be isolated from each other
  - Processing rates need to be decoupled
    - For example, physical simulations can run at higher loop rates than the rendering system

Multithreading in Game Engines

- Threads already used to some degree
  - Audio processing
    - Mixing and streaming can use a lot of time
  - Networking
    - Only awake to handle socket traffic
  - OpenGL driver usually multithreaded now
    - Normally one more thread, decoupled from main thread
  - OS will spawn threads to handle infrequent tasks
    - DirectSound, DirectInput, etc.
    - These threads almost always asleep

Multithreading in Game Engines

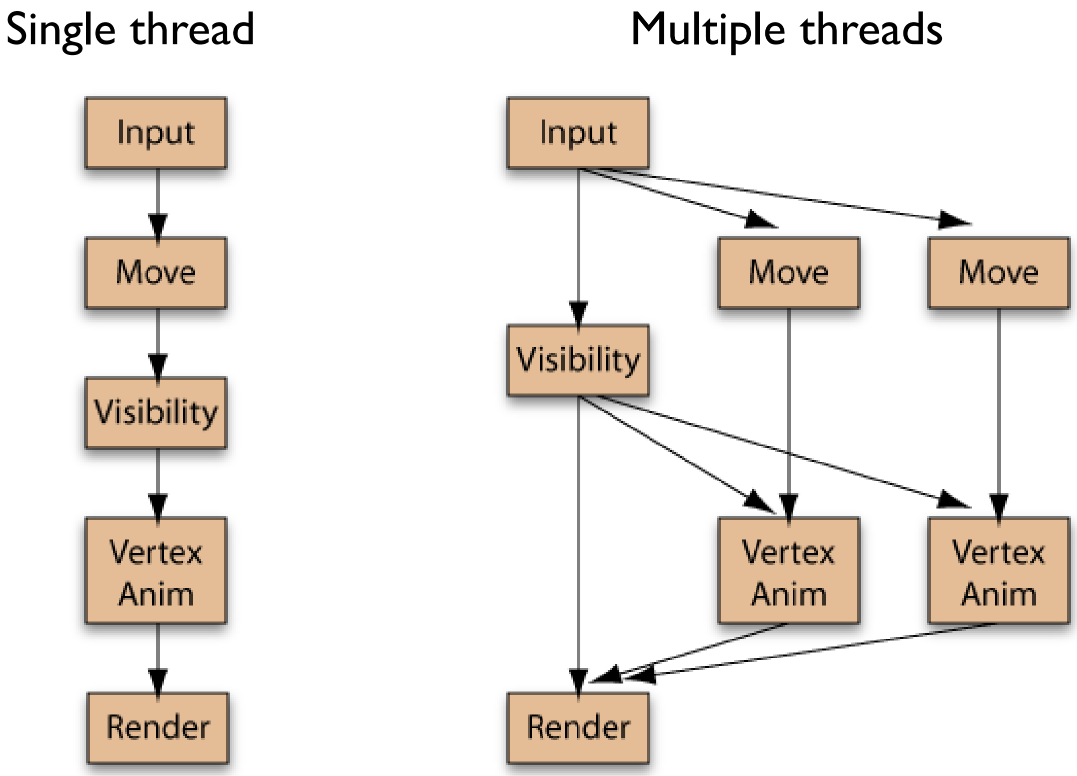
- Hard to keep more than two cores fed in current generation of game engines
- But plenty of ideas for using up the processing power
  - Physics
    - Collision detection
    - Fluid simulation
    - Cloth simulation
  - Particle systems
  - Character animation
    - Bone transformation
    - Mesh skinning

Multithreading in Game Engines

- Multithreading creates new synchronization issues
  - Many threads need to complete their tasks before rendering can occur
  - Easier to manage on consoles because applications have greater control over GPU

Multithreading in Game Engines

- One worker thread runs per processor
  - Processes jobs that are queued by main thread



Conclusions

- Nowadays, a game engine is a continuously evolving project
  - Rare to see an engine started from scratch for new games if an engine used for a previous game already exists
- Highly parallel threading is the future
  - Both on CPU and GPU
  - New engines need to be ready for this
  - Existing engines need to be modified for this

Questions?

- lengyel@terathon.com