Overview

- Modern Engine Features
- Modern Engine Challenges
- Scene Management
  - Culling & Batching
- Geometry Management
  - Collision Structures
- Shader Systems
- Example Engine Design
Modern Graphics Engine Features

- High polygon count for added visual complexity
  - Not just to make things ‘smoother’

- Some form of bump mapping for more surface detail
  - From single-light dot3
  - To general diffuse / specular / aniso per-pixel lighting

- Some form of shadows
  - From simple blobby discs under characters
  - To full shadow map or shadow volume for each light
More Features

- Particle system for splashes, sparks, etc.
- Decal System for blood, scorch marks, etc.

Performance & Visual Scalability
- Game should look good on the newer cards
  - 1280x1024 x 4X AA + 4X Aniso
- Game should look ‘ok’ on the older cards
  - 800x600 x 2X AA
- And run well on both at the appropriate resolution and Anti-Aliasing settings
Challenges

To get achieve visually rich scenes, there must be several visually interesting objects.

To get acceptable frame rates, the number of draw calls in a frame should be low:

- < 500 per frame for good frame rates

This is a CPU limitation:

- The API & Driver must do a little work every time you make a render call to draw something
- Many calls doing a little CPU work add up to a lot of CPU work
Challenges

Complexity
- Modern engines are able to lose some complexity compared to engines a few years ago
  - Software Transform
  - Software Rasterization

But, there are plenty of new things to worry about
- High poly-count worlds in low memory
- Realistic characters & animation
- Shader Management
- High Framerates
Scene Management

There are about 5 different game engine sections that need access to the geometry in the scene.

- Culling
- Rendering
- Collision
- Decals
- AI
Scene Management

- Culling – View Frustum Culling
  - Also from light’s point of view for some shadow approaches
- Rendering
  - May need to render from multiple points of view for radar, shadows, etc.
- Collision
  - May be simplified version of the rendered geometry
- Decals
  - If these are done by re-rendering scene triangles, need per-triangle collision
- AI
  - The computer needs some spatial awareness
  - For path-finding, tactical understanding, etc.
Scene Management - Culling

Goal: To quickly identify groups of triangles that can be culled out efficiently
- Typically inside a bounding volume
  - BSP Leaf, Sphere, Bounding Box

There is a tradeoff between culling efficiency and CPU efficiency:
- The ultimate culling efficiency would cull each triangle individually
- The ultimate CPU efficiency would draw the entire world in one draw call

The trick is to group most of your scene in large, easy-to-cull chunks
Scene Management - Culling

- In this scene, a world section is broken into a grid with ~300 triangle cells

- Highlighted area represents one such 3D Cell

- Probably too few tris for CPU batch efficiency
Scene Management - Culling

- Make bounding boxes too small, and clipping creates many extra triangles & vertices

- Make bounding boxes too large, and you end up sending down too much off-screen geometry
  - Can also create per-material AABoxes

**Instanced Geometry**

- Store a Axis-Aligned Bounding box, AA Cylinder or Sphere for each Instance for culling
- Don’t cull individual bone groups except for very expensive and close-to-camera characters
Scene Management - Culling

**Particle Systems**
- Store a bounding volume for each group of particles
- Cull entire group as a unit
- Also try to draw as a unit for efficiency
- If particles don’t affect gameplay, can also avoid calculating off-screen systems
Scene Management - Decals

There are several popular approaches for creating decals for bullet holes, scorch marks, blood drops, etc.

One approach renders little pieces of geometry to represent the bullet hole, etc.

- **Upside**: Low fillrate for small decals
- **Downside**: Needs to be clipped so that it doesn’t hang over a corner
- **Downside**: May Z fight with geometry, need bias
Another method uses texture mapping to apply the decal by re-rendering scene polygons with the texture applied.

- **Upside**: No need to clip to corners
- **Upside**: No depth fighting if you use the exact same geometry as used for rendering
- **Downside**: Large polygons cost fillrate for many decals
Scene Management - Decals

Either approach requires finding the exact triangles the decal touches
- Either for clipping the geometry decals to the scene geometry
- Or re-rendering them with the decal texture

Therefore, the engine must support being able to quickly find a group of nearby polygons on which to apply the decal

This has implications for the collision system...
Scene Management - Decals

- Highlighted area indicates triangles possibly covered by decal shadow.

- Amount of extra fillrate burned is more for less-tessellated geometry.

- So, more vertices can save fillrate on decals.
Scene Management - Collision

- Efficient collision with world & mesh data is a challenge in a modern engine
  - Many more polygons for required visual richness

- Standard BSP approaches won’t cut it
  - Only for very simple walls & floors will a leaf-based BSP suffice
  - The more polygons in the scene, the greater the penalty for splits
Scene Management - Collision

- One of the main problems with a standard BSP or KD-Tree (axis aligned BSP) is the depth of the tree.

- Consider every time you follow a pointer, you can assume a CPU cache miss.
  - The deeper your tree, the more cache misses you will take.
  - Cache misses can be more expensive than intersection tests.
  - Therefore, shallower tree types will perform better on high-polygon-count scenes.

- Two ways to make a tree shallow:
  - High Branching Factor – QuadTree, Octree
    - More children per node
  - Store multiple items in one Leaf.
Scene Management - Collision

This KD-Tree or BSP has 2 levels, the leftmost root and the rightmost children.

The Quadtree only also needs 2 levels for this scene.
Scene Management - Collision

Of course, if the scene is arranged differently, the KD tree or BSP tree can cope better by adjusting where the split planes go.

Standard Quadtrees and Octrees don’t do this, so require more levels. Variations with rectilinear cells, as on the right, can cope better.
Scene Management - Collision

An alternative is the Axis-Aligned Bounding Box Tree
- Good example in Game Programming Gems 2
- Also Short Tutorial on FlipCode

This Tree contains a hierarchy of AA Bounding Boxes which contain all of the geometry

The AABox Tree is not meant to represent empty space like a grid, but instead to just tightly contain the triangles
Scene Management - Collision

The basic idea is to somehow divide the # of triangles in a node in half at each step, but without clipping them to the Split Plane.

- Root Node
- Dashed line is Split Plane
Scene Management - Collision

- The triangle centroid is compared to the ‘Split Plane’
- This way each triangle only lives in one node
- No clipping to increase polycounts
  - Important for collision more than for rendering

- Left Child in Blue
- Right Child in Red
- Dots are Triangle Centroids
Scene Management - Collision

- Each node in tree contains a Axis-Aligned Bounding box, and two children
- Each child may be a Node or a Leaf
- Leafs contain the triangle data or triangle ids
- Can create tree down to individual triangle level
  - Requires compression of nodes & bounding boxes to avoid too much memory – see GPG2
- Alternatively create down to a small # of triangles per leaf, like 8 – 20
  - All triangles in leaf will be nearby in memory
  - Argues against storing tri ids, and just vertex indices
Scene Management - Collision

- The ‘Split Plane’ must be intelligently chosen to create a nicely balanced tree

- One approach is to create the AABB tree top-down

  - Create a parent node and find the AABox containing all triangles
  - Split the node somehow into two children
  - Each child gets some of the triangles
  - Each child’s AABox may overlap its sibling
  - Recurse into each child until
    - The # of triangles is small enough
    - Or the volume of the AABox is small enough
How To A Node Split into 2 Children?

A good approach is to pick the largest axis of the AABox containing all triangles in the parent node.

Then sort the triangles by their centroid with respect to the AABox’s largest axis.

- A tall AABox would have its triangles sorted by centroid.y

Now you can go through exactly half the triangles in the sorted list and give them to the left child, and the assign the rest to the right child.

This gives a median distribution, which guarantees a $O(\log n)$ search time.
Scene Management - Collision

- Modern engines will increasingly use non-splitting, looser trees with larger numbers of triangles per leaf.

- Looser trees, like AABB Trees, and loose Octrees don’t split, so they don’t increase collision polys needlessly.

- A dozen or so triangles per leaf reduce cache misses and amortize the memory cost of the bounding box and node information.
The AI also needs some view of the scene.

But it should be probably be separate from the rendering & collision views of the world.

Should probably a simpler, more symbolic view of the world than a collision structure.

The AI will use raycasting and other spatial queries, so this should be fast.

- Line Of Sight
- Enemies In Range
Scene Management - Rendering

Question: What is the most expensive render state change?
Scene Management - Rendering

Question: What is the most expensive render state change?

Answer: The one that caused you to make more draw calls.
Scene Management - Rendering

Question: What is the most expensive render state change?

Answer: The one that caused you to make more draw calls.

In general, sort by the item with the most useful coherency.
Scene Management - Rendering

You can use the GPU to reduce the number of draw calls needed for your scene.

Use per-vertex data to encode shading parameters
- Reduces need to set vertex shader constants
- Reduces need to switch vertex shaders

Example: Indexed Palette Skinning

Idea: Apply per-vertex index to other things like lighting, occlusion, etc.
Scene Management - Rendering

- Use textures to encode shading parameters
  - Reduces need to set pixel shader constants
  - Reduces need to switch pixel shaders

- Example: Put gloss into the alpha of your normal map, instead of setting it via SetPixelShaderConstant()

- Idea: Encode 4 light occlusion terms into a lightmap, and draw all 4 shadowed lights in one pass
Lighting and Shadows

Your choices for Lighting and Shadowing will largely dictate the look and speed of your game.

How dynamic is your lighting?

- **Totally Static**
  - Precompute per-vertex or lightmaps

- **Partially Dynamic**
  - Lights can change color & intensity, but can’t move
  - Build per-light occlusion term into vertex or texture

- **Totally Dynamic**
  - Perform plenty of CPU raycasting for shadowing
  - Use GPU-assisted shadows like shadow maps or shadow volumes
# Lighting Tradeoffs

<table>
<thead>
<tr>
<th>Technique</th>
<th>CPU Cost</th>
<th>Vertex Cost</th>
<th>Pixel Cost</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Lightmaps</td>
<td>Low, if using texture pages</td>
<td>Low</td>
<td>Low</td>
<td>Any # of lights and shadows free</td>
</tr>
<tr>
<td>Dynamic Lightmaps</td>
<td>High, at least when light changes</td>
<td>Low</td>
<td>Low</td>
<td>More lights cost during updates</td>
</tr>
<tr>
<td>Dynamic Lightmaps w/ Shadows</td>
<td>Prohibitive</td>
<td>Low</td>
<td>Low</td>
<td>Too many raycasts for CPU</td>
</tr>
</tbody>
</table>
# Lighting Tradeoffs

<table>
<thead>
<tr>
<th>Technique</th>
<th>CPU Cost</th>
<th>Vertex Cost</th>
<th>Pixel Cost</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occlusion Maps</td>
<td>Low, if using texture pages</td>
<td>Low</td>
<td>Medium</td>
<td>Limits # of lights to 4 or so</td>
</tr>
<tr>
<td>Per-Vertex Occlusion</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Lights Can only change color</td>
</tr>
<tr>
<td>Stencil Shadows on CPU</td>
<td>High, for batch count and silhouettes</td>
<td>Low</td>
<td>High</td>
<td>Limited to 3 lights per surface</td>
</tr>
</tbody>
</table>
## Lighting Tradeoffs

<table>
<thead>
<tr>
<th>Technique</th>
<th>CPU Cost</th>
<th>Vertex Cost</th>
<th>Pixel Cost</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil Shadows on GPU</td>
<td>Medium for batch size</td>
<td>Very High</td>
<td>High</td>
<td>Limited to 3 lights per surface</td>
</tr>
<tr>
<td>Depth Shadow Maps</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Aliasing Artifacts</td>
</tr>
<tr>
<td>PRT via Spherical Harmonics</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Only infinite lights &amp; no animation</td>
</tr>
</tbody>
</table>
Shader Management

There are two main ways to handle shaders, depending on your type of game

Open-Ended - Artist-driven from within the level editor

- Highly Flexible
- Use HLSL / .FX files to manage complexity
- Somewhat Complex to support many shader types
  - Use Annotations to identify shader parameters
- Can create explosion of shaders if not careful
- Switching shaders often is not good for reducing draw calls
Shader Management

- Unified Shader Model – Driven from the Engine Capabilities and/or Game Needs
  - Fewer, more specific, optimized shaders
  - Practical to do C++ coding to set up shaders
    - Can still use .fx files, but not needed as much
  - Shaders are from a more limited set of choices
  - Good for higher framerates by limiting maximum # of draw calls due to shader changes
  - Must build shader parameters into geometry & textures to get the speed benefit
Questions So Far?
Test Engine Overview
Top level of the scene is a 3d Grid of 16x16x16 meter cells

- Triangles are clipped to the grid
- Each Cell has a Vertex and an Index Buffer
- AABBTree of collision triangles matching the tessellation of the rendering triangles

- Also a vector of material records
  - Contains Index Buffer range for triangles
  - Contains AABox for triangles with material

List of Moving Entities

- Contains AABox
- Contains reference to mesh data for rendering only
Test Engine - Overview

- Advantage to breaking the world into large cells
  - Efficient for culling
  - Can share same VB and IB without going over 65K vertex or triangle limits
    - Can use 16-bit indices for IB
    - Can use 16-bit indices for AABB tree
  - Can compress AABB boxes in tree to 16 or 8 byte per axis and still have good precision
  - Can more quickly reject moving entities in other cells
  - Can restrict lighting to only 7 lights per tile
Many Features, Few Draw Calls

- An entire world cell is drawn in one draw call
  - Up to 7 Lights
  - Diffuse & Specular Bump Mapping
  - Soft shadows
  - Gloss-Mapped, Color Shifted Specular
  - Masked Emissive
  - Water or Fog Depth stored in Dest Alpha

- Fog, Mist and Water are a partial alpha pass
  - Blend in fog layer colors based on dest alpha
Shadows & Deep Water
Shadows for world geometry are pre-calculated for up to 7 lights

Per-Light occlusion terms are stored in diffuse rgba and specular rgb per vertex

The vertex shader calculates the 7 L vectors
  Scales the L vectors down by the occlusion & attenuation terms
  Performs per-vertex N.L
  Adds up scaled L vectors
Averaged L Bump Mapping

The Pixel Shader uses this averaged L vector to perform bump mapping – ‘Averaged L Bump Mapping’

Bump mapping is nice, but not worth it to do for many lights

This way, if lights change intensity or turn on and off, the bumps respond to the most intense lights

The bump mapping still corresponds to the scene’s lighting, but no need to do up to 7 rendering passes for 7 lights
Averaged L Vectors for L Bump Mapping

“Averaged L Bump Mapping”
Metallic Specular
Questions?

sdiétrich@nvidia.com

simmer@spies.net