CS101c
GPU Programming

Luke Durant, Russell McClellan, Tamas Szalay

Today

- Shadows
- GLSL debugging
- GPU Clipmap Terrains

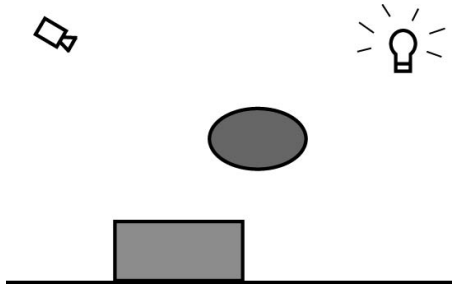
Shadows

- Parallax mapping shadows used often in games
- Not the general way to do shadows, because it's too closely tied in with parallax mapping.
- Many ways to do shadows
  - Raytracing!
    - Everyone wants Real-Time Raytracing …talk to nvidia
  - Shadow Maps
  - Volumetric Shadows
  - Shadow baking
  - Simpler things…

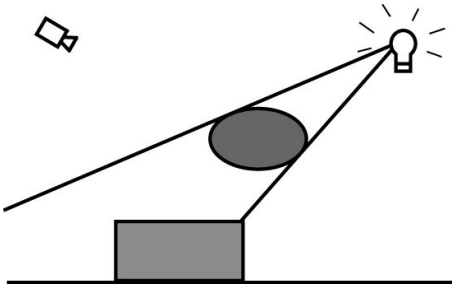
Why Shadows?

- Realism
- Helps reveal shape and form
- Define spatial relationships between objects
- Actual game mechanism?

Definitions of 'In Shadow'



Definitions of 'In Shadow"



Two definitions

- Look at scene from light's perspective
- If there's something closer than a given surface, it's in shadow.
- Implementation?
  - Shadow maps
- "Throw" the back half of every object to infinity along light rays
- Enclosed surfaces in shadow
- Implementation?
  - Volumetric shadow
  - 'Carmack's reverse'

Primitive Shadows

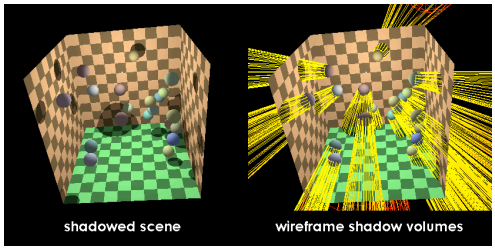
- Define "entities"
- For each entity, choose a radius
- Cast a ray down, picking a triangle
- Solve for overlap with adjacent triangles
- Draw decal with low alpha-value

Better

Stencil Buffer

- Another buffer, similar to depth buffer.
- Set integer values per-pixel
  - Usually only 1 byte per-pixel
- Can be used to restrict what's rendered
- Different use for volumetric shadows.

Volumetric Shadows



Volumetric Shadows+Stencil Buffer

- Check if something is inside shadow volume.
- Render it, setting depth value.
- If it's a front facing (to light) surface, increment corresponding stencil buffer pixel.
- Decrement stencil buffer pixel for back facing
- Check for stencil value != 0 (provided we don't ever saturate, or have ridiculous nested shells).

Principle behind Shadow Mapping

- Render the scene from the light's point of view, and save the depth buffer.
- Render the scene from the camera's point of view. Shadow condition:

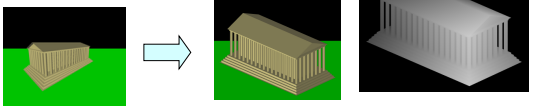
distance from fragment to the light

>

value of fragment in depth buffer

Step 1: Create Shadow Map

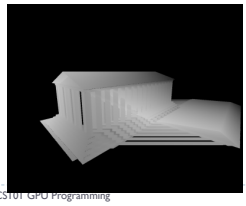
- Render the scene from light's point of view.
- Perspective projection for spotlight
- Orthographic projection for directional light (sun)
- Save the depth buffer.



Step 2: Depth map projection

- Draw the scene from the camera viewpoint
- To do depth comparison, must compute coordinate of point as seen from the light.

Visualization of depth map projected onto the scene



Light Space calculation

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} * M * V\_light * P\_light * [x,y,z,w] = [s,t,r,q]$$

[-1,1] → [0,1]

scale and bias matrix, translates clip space coordinates to texture index coordinates

modeling matrix

object space → world space

projection matrix

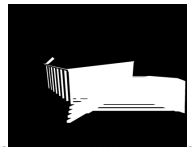
view space → projection space for view from light source

view matrix

world space → light at origin, looking at world space origin

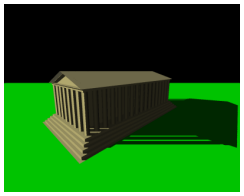
Step 3: Depth map test

- Get [s/q, t/q, r/q, 1] from previous step.
- s/q, t/q = depth map lookup coordinates
- Compare r/q to depth map entry
- If r/q > depth map entry, point recieves no contribution from light source.



Step 4: Drawing the Scene

- Draw the scene from the camera's point of view.
- Do depth testing in the fragment shader

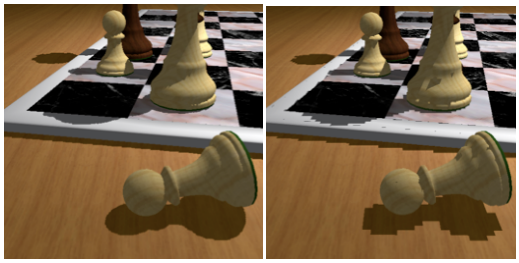


GLSL Support

- Shadow maps natively supported
- uniform sampler2DShadow map_name;
- vec4 shadow2Dproj(map_name, vec4 strq coords)
- vec4 shadow2D(map_name, vec3 str_coords)
- Compares z coord with shadow map entry
- To use: shadow2Dproj(…).x = 0 if shadow, 1 otherwise
- gl_TextureMatrix

Problems: Z-Fighting

Problems: Aliasing



GLSL Debugging

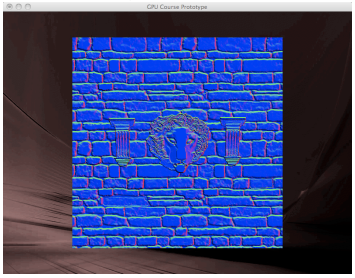
- Not a lot you can do.
  - Visual debugging!
- There are some random tools; none are known to be particularly useful
- Mostly just hack things to see what data you're interested in.
- Need a to know a value in the vertex shader?
  - Either set vertex positions to those values, and render points instead of triangles.
  - Or vary values to frag shader, and do debug there

GLSL Debugging

- Frag shader debugging
  - Set output color!
- I.e., can't figure out what lighting is doing, suspect normals are wrong.
  - gl_FragColor = vec4(normal, 1.);
  - But you have to be careful with coordinate systems; make sure you know what colors correspond to what values in what coordinate system.

GLSL Debugging

- Normals for lion, in surface coordinates:



GLSL Debugging

- Need to check numeric values?
  - Render to texture
  - glReadPixels
- Can change scene rendered
- Render full screen quad to focus on specific values
  - Read these pixels back out
  - Getting into GPGPU techniques through GLSL

GPU Terrains

- So far, mostly done things in fragment shader
- A lot can be done in vertex shader
  - At least since texture support was added in vertex shaders
- Common application is 2.5D terrains
  - I.e., Terrains that can be represented with a heightmap
  - No overhangs, etc.

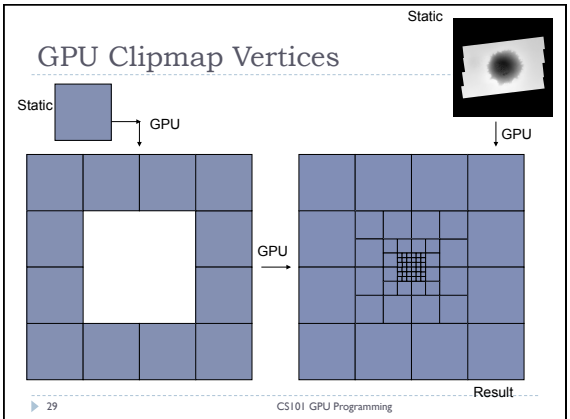
GPU Terrains

- What do you do if you have a large dataset?
- 2048*2048 heightmap gives ~8.5 million triangles to render all of the data.
- May need all of the data in certain regions of interests; i.e., wherever the camera is closest
- Don't care about data further away as much, it's just slowing the rendering down.

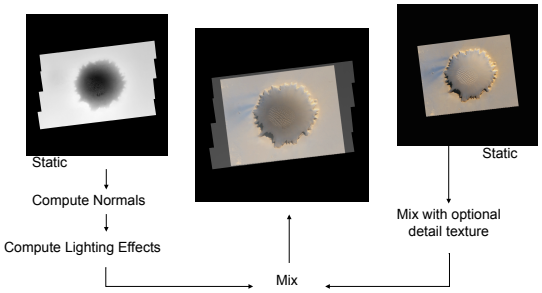
GPU Clipmapping

- Idea: Render high resolution near the camera, have the terrain rendered with less detail further away.
- Regular grids easiest, just have to be careful to make sure seams don't appear.
- Because we can transform vertices on the GPU, we can repeatedly pass one square grid of vertices into shader with different uniforms, then scale and translate to get varied vertex resolution.

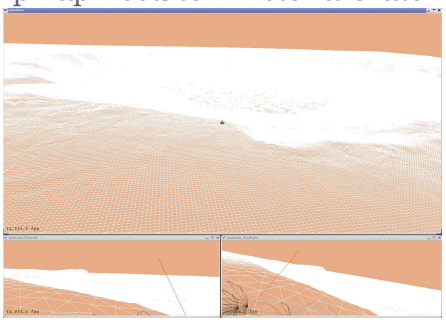
GPU Clipmap Vertices



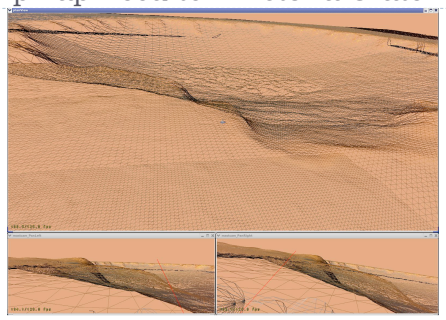
GPU Clipmap Fragments



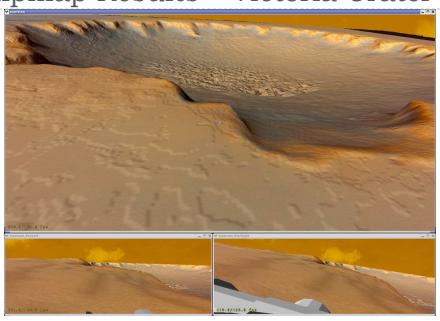
Clipmap Results – Victoria Crater



Clipmap Results – Victoria Crater



Clipmap Results – Victoria Crater



Clipmap benefits

- Fast!
  - 200fps faster on 2048x2048 Victoria Crater
- Can generalize to arbitrarily sized data sets
  - Use different heightmap textures on different 'ring levels of detail'.
- Less work for CPU; doesn't have to do all of the vertex computations.

Issues with clipmaps

- A little bit awkward not having geometry tied down
  - The actual vertices follow the camera, which works out everywhere insides, but if you can see the edge, it's strange to see that moving around.
- Must be very careful to avoid seams between LODs.
- For very coarse areas, the terrain can 'jump', as it goes from low detail -> high detail vertices.