Shadow Techniques

Sim Dietrich
NVIDIA Corporation
sim.dietrich@nvidia.com
Lighting & Shadows

• The shadowing solution you choose can greatly influence the engine decisions you make

• This talk will outline some of the more popular shadow techniques and compare and contrast them
One very key observation that changes the way realtime lighting is implemented is whether to:

1) Start with a lit scene and subtract or modulate out light in shadowed areas

2) Start with a black scene and add in light contributions for each light in the scene, except where there is a shadow

Most current engines do the 1st, but the 2nd is what engines will be doing going forward
Most games today lay down gray textures or polygons to indicate shadows. This breaks down for > 1 light, and doesn’t produce correct colors.

An engine with > 1 light touching a pixel would do better to just filter out lights based on shadow occlusion.
Many current engines compute something like this:

\[ FB = \text{DiffuseTex0} \times (\text{Light0} + \text{Light1} + \text{Light2} \ldots) \]

They then indicate the pixel is in shadow with respect to Light0 like so:

\[ FB *= 0.75 \]

Obviously this is wrong, because Lights 1 & 2 get darkened too.
Shadows

- The more correct way to go is like so:

  \[ FB = \text{DiffuseTex0} \times (\text{Light0} \times \text{Mask0} + \text{Light1} \times \text{Mask1} + \text{Light2} \times \text{Mask2} \ldots) \]

- The Mask Values are
  - 0 if in shadow with respect to that light
  - 1 if in the light
  - Some value in-between on a soft shadow edge
Another Problem with Shadows via Darkening

- If your shadowing approach shadows after lighting, you need to worry about shadow leakage.
- Your shadows may extend through occluders and incorrectly shadow areas that couldn’t be lit in the first place.
Example of Problem

In this case the Tree may be darkened even though it may not have been lit by the light to begin with.
Render with Light, not Shadow

- The solution is to render with illumination for each pixel, except where there is a shadow.
- This avoids shadow leakage.
- Also it allows for any # of lights to be rendered with properly colored shadows.
Adding Light to a Dark Framebuffer

- To accumulate each light’s contribution to the scene, use additive alpha blending like
  \[ \text{SRCCOLOR} \times \text{ONE} + \text{DESTCOLOR} \times \text{ONE} \]
- To ensure that the scene is rendered correctly, you have to clear the color buffer and set the Z buffer first
- This is to prevent a nearby spotlight that overlaps a distant occluded spotlight from adding in the occluded spotlight’s color and intensity
Setting the Z Buffer

- The simplest way to set the Z buffer is to simply render the entire scene with only black, or with ambient lights and shadows
  - Precalculated static shadow maps can be included during this pass
- It may be faster, depending on geometric complexity, to do Z by itself first, and then lay down the rest of the scene with the Z test set to `D3DCMP_EQUAL`
- One advantage of this architecture is that you get the maximum Z occlusion benefit without sorting your scene from front to back
Storing the Masks

- There are three main ways to apply the shadow mask once it has been generated
  - Use Alpha Test to throw away pixels that aren’t lit
  - Use Stencil Test to throw away pixels that aren’t lit
  - Use blending, either DestAlpha or texture blending to zero out illumination terms that shouldn’t contribute to the scene due to shadows
How to Generate Shadow Masks?

• The next question is how to generate the shadow masks to begin with:

• Analytical Approaches
  • Shadow Volumes
  • Poly Projection

• Pixel-Based Approaches
  • Render a Depth-Incorrect Soft Shadow
  • Use Depth-Correct Shadow Maps
  • ObjectID / Priority Buffers

• Hybrid Approaches
Analytical Shadows – Shadow Volumes

- Two main ways to generate Shadow Volumes analytically:
  - Add new polys representing the volume extruded away from the light
    - Always works
    - CPU intensive
    - Can’t do in a vertex shader
    - Requires CPU to perform animation
  - Stretch existing “back facing” vertices away from the light
    - Cheaper, can be done in a vertex shader
    - Like the “Motion Blur” Tiger Demo
    - Only works for well-tessellated convex models
Extruding Objects For Shadow Volumes

Original Object

Extruded Object
Analytical Shadows - Shadow Volumes

- Once Shadow Volumes are computed, they are rendered into the scene using the Stencil Buffer.

- One way to render shadow volumes involves updating Stencil depending on the depth test passing or failing.

- This particular technique avoids having to cap your shadow volumes at the near clip plane if the camera is inside the volume.

- At the cost of doing each light’s shadow in its own pass, and clearing stencil in between lights.
Analytical Shadows - Shadow Volumes

• For Each Light:
  • Clear the Stencil to 0
  • Render Front Faces of the shadow volume incrementing stencil on Z pass
  • Render Back Faces of the shadow volume decrementing stencil on Z pass

• If the Resulting Stencil Value for a pixel $\neq 0$, then it is in shadow
Analytical Shadows - Shadow Volumes

- If the camera is inside one or more shadow volumes, the algorithm changes
  - You must clear the stencil not to 0, but to the # of stencil front volume polygons the camera is inside
  - This compensates for the front facing volume sections that will be clipped by the near clip plane
  - We still need to clip the back faces of shadow volumes to the far clip plane
Analytical Shadows – Shadow Volumes

Case 1: Object in the Shadow Volume

Marks Z Fail Area

The Front face makes Stencil 1, the Back Face does nothing

Occluder

Viewer

Shadow Volume

Front Face

Back Face
Analytical Shadows – Shadow Volumes

Case 2: Object in Front of the Shadow Volume

 Marks Z Fail Area

There are two Z fails, so Stencil is Zero

Occluder

Viewer

Shadow Volume Front Face

Shadow Volume Back Face
Analytical Shadows – Shadow Volumes

Case 3: Object in Back of the Shadow Volume

- Marks Z Fail Area
- There are no Z fails, so Stencil is Zero

Occluder

Viewer

- Shadow Volume Front Face
- Shadow Volume Back Face

There are no Z fails, so Stencil is Zero.
Analytical Shadows – Shadow Volumes

- Now that the stencil is marked, we can use the Stencil test as our Light Occlusion Mask
  - If Stencil $== 0$, Render Light to Frame Buffer
  - If Stencil $!= 0$, Don’t Render Light to Frame Buffer
Analytical Shadows – Shadow Volumes

- **Advantages:**
  - No aliasing problems

- **Disadvantages:**
  - Usually requires CPU work on the geometry to generate the silhouette
    - Performance is bounded by geometric complexity of shadow casters
  - However, the stretching variation can be done on the GPU if your meshes are well-tessellated and have smooth normals
  - Must use 32 bit Z (for stencil)
  - Highly variable fill requirements
    - Can slow down if the camera is near a shadow volume
The other analytical approach is to actually project each vertex of the shadow caster onto the shadow receiver.

- This requires generating shadow polygons on the CPU.
- One must clip them as necessary to correctly fall over the receiving geometry.
  - This avoids shadows hanging over cliffs, etc.

This requires either:

- Finding the minimal silhouette of the object.
  - No overlapping polygons allowed.
- Or ensuring that double-hits don’t double darken.
  - This can be done by either using additive blending with Destination Alpha or using Stencil operations.
Analytical Shadows – Shadow Caster Projections

- **Advantages:**
  - No aliasing problems
  - Fill requirements typically not as intense as shadow volumes, because we are drawing flat triangles and not triangles spanning a volume

- **Disadvantages:**
  - Always requires CPU work on the geometry to generate the silhouette and to clip to receiving geometry
    - Performance is bounded by geometric complexity of shadow casters AND shadow receivers
  - Must either clip original geometry to match generated shadow polygons, or draw the generated polys with Z bias
  - If not doing minimal silhouette extraction on the CPU, one must use 32 bit Z (for stencil) or 32 bit Color (for dest alpha) to avoid double darkening
  - Hard edged shadows
  - Requires Z bias
Pixel – Based Shadows: Depth-Incorrect Soft Shadows

- This technique involves a render-to-texture of the shadow casters
  
  - Clear the texture to white
  
  - Render in the shadow casters in black
  
  - When you draw the shadow receivers, project them onto the shadow texture for the light masking term
  
  - These are easy to make softer around the edges
    
    - Use a low-res shadow texture
    
    - Use bilinear filtering to soften the edges
    
    - On multitexture hardware, take multiple jittered bilinear samples to smooth out the edges even more
Pixel – Based Shadows : Depth-
Incorrect Soft Shadows
Pixel – Based Shadows: Depth-Incorrect Soft Shadows

- The main problem with this approach is dealing with the shadow casters themselves.
- If you treat the shadow casters as receivers also, they will be shadowed even if they are the closest thing to the light.
- If you don’t allow shadow casters to receive their own shadow, then they don’t get self-shadowing.
  - So, their arm can’t shadow their chest.
  - This is OK for convex objects, but not for complex characters.
Pixel – Based Shadows: Depth-Incorrect Soft Shadows

- Another problem with this approach is that it is difficult to combine multiple shadow casters into the same render-to-texture.

- The reason is that the resulting texture is just black and white, there is no distinguishing one object from another.

- For this reason, you really need a separate shadow texture for each pair of casters/receivers.

- This technique, therefore, is often limited to project a soft shadow of a character onto a terrain or floor.
Pixel – Based Shadows : Depth-Incorrect Soft Shadows

- Advantages:
  - Simple
  - Only have to render one object to a texture
  - Soft edges, although they can be jaggy

- Disadvantages:
  - Not a general solution
  - Shadow casters don’t self shadow
  - Doesn’t work for objects that are both shadow casters and receivers
Instead of rendering black to a white texture to indicate the presence or absence of a shadow caster, what if we had a Shadow Depth Buffer instead?

The idea is to identify the depth of the closest pixel to the light:

- This is accomplished via a render-to-texture operation.

If when rendering the scene we check to see if the current pixel’s distance from the light is greater than the closest pixel that we stored in the depth buffer.

- If so, the pixel is in shadow.
Pixel – Based Shadows : Shadow Depth Buffers
Pixel – Based Shadows : Shadow Depth Buffers

- Rendering to the Shadow Depth Buffer
  - Set up your view matrix to be the light’s “LookAt” matrix
  - Set up the projection matrix based on the light type
    - For spotlights, use the penumbra angle for the FOV
    - For directional lights, use an orthographic projection
    - For point lights, use a cubemap
      - And render once for each face with a 90 degree FOV
  - Render your depth value into the texture
    - As an Alpha or Color Value
      - 0 means at the light plane
      - FF means at the edge of the light’s range
Pixel – Based Shadows : Shadow Depth Buffers

- Where do we get the value to write into the depth buffer?
  - Typically this is a texgen operation, with the texture coordinates corresponding to [0..1] along the Z axis of the light’s view matrix.

- Texture coordinate 0.0 corresponds to at the light plane, and 1.0 would be at the edge of influence of the light.

- The texture contains a ramp for the alpha or color value.

- Alternately, we can use a vertex shader to compute the distance from the light plane.
  - Watch out for clamping problems.
Pixel – Based Shadows : Shadow Depth Buffers

- Then when rendering your scene:
  - For each object that crosses the light’s area of influence
    - Typically a bounding sphere or a frustum
  - Calculate its depth value exactly as you did when you created the shadow depth buffer
  - Also project the pixel onto the shadow depth buffer
  - Subtract the Calculated Depth from the Projected Depth
    - If the result is 0, the pixel is lit
    - If the result is positive, the pixel is in shadow
To handle point-lights properly, you need to render to all 6 faces of a cubemap to get the depth from each point.

However, planar depth won’t work as it does for spotlights or directional lights.

It won’t work if each face of the cubemap contained a depth buffer storing planar depth from that face.

- There is no way to dynamically change our depth calculation on a per-pixel basis.
- One would need a volume texture for this.
Pixel – Based Shadows : Shadow Depth Buffers

- But, although we can’t use planar depth, we can use a function of spherical range or range squared instead.
- This way the shadow test can be consistent between the depth generation during the cubemap construction and the scene rendering.
- Use 2 2D textures to compute a range function such as $1 - D^2$
  - Where $D = \sqrt{x^2 + y^2 + z^2}$
- This is the Attenuation Map technique again!
- Downside is the low precision: $<< 8$ bits
Rendering 6 faces of a cubemap is not cheap, so here are some tips for speeding it up:

- If the point light doesn’t move, take a snapshot of the static geometry
  - Then save off the 6 Z buffers and the 6 shadow depth buffers that the static geometry represents
- Use these as the starting point when updating the scene
  - Render your dynamic characters and objects into copies of these buffers, this saves most of the fill requirements for keeping these updated
- If the point light moves, all bets are off
  - You might try not updating some faces of the cube each frame, but this could introduce artifacts
### Pixel – Based Shadows: Shadow Depth Buffers

- **Advantages of this approach are:**
  - Supports object self-shadowing
    - Since each pixel is treated individually, it is possible for one part of an object to shadow another part correctly

- **Disadvantages:**
  - Typically done with only 8 bits of color or alpha precision
    - Not enough for complex scenes
  - Suffers from aliasing problems
    - When shadow testing, you won’t always project exactly onto the same shadow buffer pixel, causing a closer or farther depth value to be found instead
  - Hard, jaggy edges
Pixel – Based Shadows : Shadow Depth Buffers

• Some things to do to improve quality:

• When creating the Shadow Depth Buffer, maximize precision by finding the nearest and farthest objects that may be lit by the light
  • Make the closest point on the closest object map to texture coordinate 0.0
  • Make the farthest point on the farthest object map to texture coordinate 1.0

• “Bias” your Calculated depth value when you are performing the shadow test
  • Push your calculated depth value a bit towards the light
  • This will reduce shadow aliasing artifacts
Pixel – Based Shadows : Object ID / Priority Buffers

- Priority or ObjectID buffers are similar to Shadow Depth buffers in that both are per-pixel approaches.

- ObjectID Buffers work by identifying each “Object” in the light’s range and giving it a unique numerical ID:
  - An Object is defined as something that can’t shadow itself.
  - So, any convex object or piece of a convex object will do.
Pixel – Based Shadows : Object ID / Priority Buffers

- Next each object in the light’s range has it’s ID rendered to a texture
  - After this step, the buffer contains the ID of the closest object for each pixel

- Now we setup similarly to the depth buffer technique
  - Compare the ID of the object you are drawing to the one looked up in the buffer
  - If they are the same, the pixel is lit
  - If they are different, that means there must be some other object closer, so the pixel is in shadow.
Pixel – Based Shadows : Object ID / Priority Buffers

• Some HW supports generating a unique ID for each polygon submitted

• This is more convenient, but doesn’t solve the real issue
  • Two adjacent coplanar polygons with different IDs can still alias with each other

• The only solutions are :
  • Use per-object ID’s instead of per-triangle
  • Perform multiple jittered tests and only shadow if all tests agree the pixel is in shadow
Pixel – Based Shadows: Object ID / Priority Buffers

- ObjectID buffers have another advantage over depth buffers – they work better with cubemaps for point lights.

- The “multiple projection” problem doesn’t occur because ObjectID buffers don’t store depth, just an ID.
Pixel – Based Shadows : Object ID / Priority Buffers

**Advantages of this Technique :**
- Can support any light range with equal precision
- For convex objects, it works great
- Doesn’t suffer from 8 bit precision issues like the depth buffer approach
- Works better for point lights

**Disadvantages of this Technique :**
- Objects must be convex or they won’t self-shadow
  - To handle this, you can break objects into smaller convex pieces, each with their own ID
- Suffers from aliasing problems
  - When shadow testing, you won’t always project exactly onto the same shadow buffer pixel, causing a different ID value to be found instead
- Hard, jaggy edges
Hybrid Approaches?

- There are two hybrid approaches that I am aware of:
  - Render To Texture / Shadow Volumes:
  - ObjectId & Shadow Depth Buffers
Render To Texture / Shadow Volumes

- Render an object to a texture, and then use that texture to reconstruct the shadow volume
  - But the HW stall required to do this would only be a win for extremely complex shapes
  - Introduces aliasing problems into the stencil shadow technique
    - What if part of the object is too small to show up in the render-to-texture buffer but big enough to be visible on screen?
ObjectID & Shadow Depth Buffers

- ObjectIDs are great because they work at any light range at all – good for inter-object shadowing

- Shadow Depth Buffers are great because they support self shadowing – good for intra-object shadowing

- We can combine the two to create a better and more robust approach than either alone

- The basic idea is that the shadow depth buffer contains both an ObjectID and a depth value for each pixel
  - Each object has its own ID as before, but the Shadow Depth buffer is actually computed per-object, so self-shadowing precision is maximized
Red Vertical Axis – ObjectID from 0 to ff
Green Horizontal Axis – Ramp from 0 to ff
Blue Horizontal Axis – Ramp from 0 to ff repeated 8 times – limited by max size of texture
Blue represents the low 8 bits of depth.
Green distinguishes the high 3 bits that represent the 8 wraps of blue.
ObjectID & Depth Buffer Shots

Shadow Buffer

Mapped Object

Difference
Conclusion – you really need to filter the results of multiple samples, especially if using ObjectID.

With Depth Buffers only, you can give away some low bits of precision to avoid this.
Ways to Smooth Shadow Edges

- For analytical techniques
  - Render multiple shadow volumes, increasing the light contribution each time in dest alpha

- For pixel-techniques
  - You can’t filter either Depth buffers or ObjectID buffers
  - Instead, do multiple jittered samples of the shadow buffer and accumulate the percentage in shadow
  - This can be done in a single pass on 4 texture hardware
Other Shadow Ideas - EMBM

- Use EMBM for shadows
  - Probably not the fastest method, but an interesting exercise
- All depth-buffer shadow techniques are basically an interated depth from the light subtracted from a projected depth buffer lookup

- EMBM works by performing
  - $dU, dV = \text{TextureLookup}(U, V)$;
  - $dU \times= \text{Mat2x2}$
  - $dV \times= \text{Mat2x2}$
  - $U` += dU$
  - $V` += dV$
  - $\text{FinalColor} = \text{TextureLookup}(U`, V`)$
Other Shadow Ideas - EMBM

- If we make the Mat2x2 be:
  
  \[
  \begin{bmatrix}
  -1 & 0 \\
  0 & -1
  \end{bmatrix}
  \]

  Then we get:
  
  \[
  U' = dU \\
  V' = dV
  \]

  U’ can be our iterated depth value from the light
  dU is the depth buffer sample
  Essentially the ‘Bump map’ is really a shadow map
Other Shadow Ideas - EMBM

- The final texture lookup needs to encode what to do for each result of
  - Iterated Depth – LookedUpDepth
- Use CLAMP addressing mode, and make the left-most texel white, and all others black
- This allows only the lit pixels through
  - $V'$ can be used for another jittered lookup
  - Make the result texture only shadow if both the $U'$ and $V'$ lookups agree
Other Shadow Ideas – 24 Bit ObjectID Buffers

- We can use R, G and B for 24 bits of Object ID
- Simply subtract the known ObjectID Buffer’s value from the rendered object’s ObjectID Value, then perform a dot product to sum the results

- Force the dp3 to replicate into alpha
  - sub r0, c0, t0 // Compare Known ObjectID
  - // to Object ID Buffer
  - dp3 r0, c0, c1 // c1 contains 0, 1, 1, 1

- r0.rgba now holds either zero for a match, or non-zero for in shadow
- Set alpha test to kill pixel if the it is non-zero
Questions...

Sim Dietrich
Sim.Dietrich@nvidia.com
www.nvidia.com/Developer.nsf