Chapter 10. ATI Shading

Pedro Sander

Chapter 10
Computation Culling with Explicit Early-Z and Dynamic Flow Control

Pedro V. Sander
ATI Research

John R. Isidoro
ATI Research

Jason L. Mitchell
ATI Research

Introduction

In last year's course, we covered applications that use explicit early-z culling as a form of flow control for graphics hardware. As described below, that method allowed the GPU to selective skip pixel shader execution for specific pixels, thus culling the amount of unnecessary computation.

With the advent of graphics hardware supporting dynamic flow control on the pixel shader, unnecessary computation can now also be avoided by strategically placing conditional statements in the pixel shader. In this talk we will describe the tradeoffs between using these two approaches and show some applications of each optimization.

Early-Z

Prior to execution of the pixel shader, the GPU performs a check of the interpolated z value against the z value in the z buffer. This occurs for any pixels which passed the hierarchical z test and which are actually going to use the primitive's interpolated z (rather than compute z in the pixel shader itself). This additional check provides not only an added efficiency win when using long, costly pixel shaders, but also provides a form of pixel-level control flow in specific situations. In a number of applications such as volume rendering, skin shading and fluid simulation, the z buffer can be thought of as containing condition codes governing the execution of expensive pixel shaders. Inserting inexpensive rendering passes whose only job is to appropriately set the "condition codes" for subsequent expensive rendering passes can increase the performance of several applications.

Early-Z culling also takes advantage of hierarchical-z culling that is present on current graphics hardware. So, 8x8 pixel blocks can be culled in unison by the hardware if they all fail the z test. Thus, explicit early-z culling is especially useful in circumstances where large regions of the screen have the same "condition code", whereas if the condition codes have a high-frequency checkered pattern, the optimization is not as significant.

Dynamic flow control

The ps_3_0 shader model supports dynamic flow control during shader execution. This feature allows efficient culling of computation by using conditionals in the pixel shader. One example in which such a feature is useful is for skipping all the light computation the surface is facing away from the light. Another example is a fully fogged pixel. There are several other instances in which such an optimization is useful.

In previous shading models, in which dynamic flow control was not supported, both paths of a conditional had to be followed and a lerp() instruction was used to "blend" between these results according to the result of the conditional expression. Now the GPU can selectively execute certain instructions within the shader.

Note that shaders generally execute in lock-step within a small pixel neighborhood. Dynamic branching is not always a performance win and can have some inefficiencies, especially if different pixels take different paths. Also, oftentimes it is cheaper for the hardware to take both paths rather than execute the branch even if all pixels take the same path. The HLSL compiler analyses the tradeoffs between emitting an algebraic expression versus using dynamic flow control and emits the one it expects to be cheapest.

Tradeoffs between early-z and dynamic flow control

Early-Z Culling:

- **Cost:** Oftentimes requires an additional pass to populate z-buffer with the condition codes. It also needs to store any necessary program state in auxiliary buffers.
- **Benefit:** The shader execution for a particular set of pixels is completely culled where desired.
- **Considerations:**
    - Takes advantage of the hierarchical z-buffer to cull larger blocks.
    - The z-buffer not available for other computations.
    - Useful for screen-space processing on a subset of the pixels.

Dynamic Flow Control:

- **Cost:** The branching instruction.
- **Benefit:** Certain code paths are culled; efficiency depends on whether different branches are taken for the same set of pixels.
- **Considerations:**
    - Single pass over geometry. Early-z could require multiple passes unless deferred shading is employed (which might require MRT).
    - Z-buffer is available to store depth.

To better illustrate these distinctions, we next present some examples of situations in which computation was culled by either early-z or dynamic flow control.

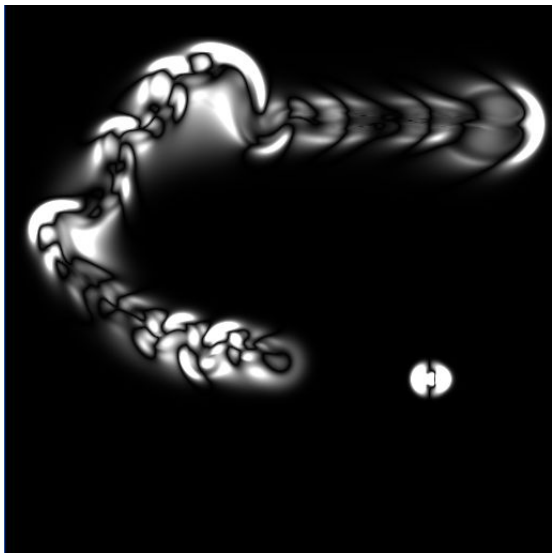
Early-Z: Fluid flow simulation

Next we will describe an application of explicit early-z culling for fluid flow simulation. As we will see this is a clear example where early-z culling is a better option.

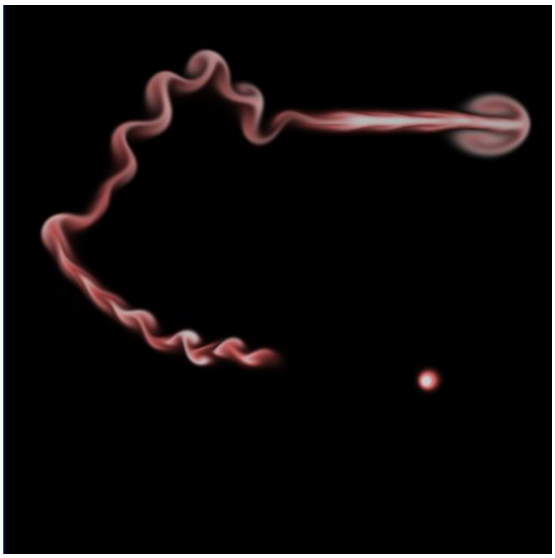
As we have seen in the past few years at SIGGRAPH and other conferences, graphics processors are being applied to broader areas of computation, such as simulation. Due to their highly parallel nature and increasingly general computational models, GPUs are well matched with the demands of fluid simulation. We have implemented a Navier-Stokes fluid simulation on the GPU, including the use of early-z as a means of avoiding certain unnecessary computations, speeding up our simulations in some cases by a factor of three [Sander04].

Our technique is based on the observation that, in many applications, flow is often concentrated on certain regions of the simulation grid. The early-z optimizations that we employ significantly reduce the amount of computation on regions that have little or no flow, saving the computational resources for regions with higher flow concentration, or for other objects in the rendered scene.

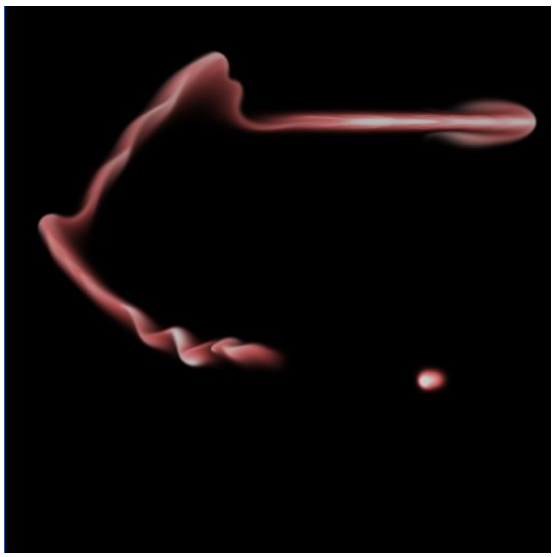
Our optimization is performed during the most expensive step of the simulation: the projection step. This step is performed as a series of rendering passes to solve a linear system. We use a relaxation method for this step. The higher the number of iterations (rendering passes), the more accurate the result is. Instead of performing the same number of passes on all cells, we perform more passes on regions where the pressure is higher and fewer passes on regions with little or no density. This is accomplished by performing an additional inexpensive render pass that sets the z value of each cell in the simulation based on the maximum current value of that cell and its neighbors from the pressure buffer of the previous iteration of the simulation. Then, we set the depth compare state to LEQUAL and linearly increase the z value on each of the projection render passes. On the first pass, all cells are processed, and on the subsequent passes, the number of cells that are processed gradually decreases. Figure 1b shows the pressure buffer which is used to set the z buffer that cull the projection computation. Darker values indicate regions of lower pressure, where fewer iterations need to be performed.



(a) Pressure buffer for simulation culling



(b) Early-Z



(c) Brute force

Figure 1 – Fluid flow Early-Z optimizations and comparison with Brute force

The choice to use early-z culling for this optimization is based on the fact that all operations are done in image space, and thus the z-buffer is not being used for depth. More importantly, the conditional decision results in either processing or completely culling a particular shader instance, as opposed to culling parts of a shader. Early-z culling completely prevents the pixel from being executed.

This culling technique is also suitable for fluid flow simulations with blockers. Since no computation needs to be performed on most cells that are blocked (e.g., white cells in Figure 2a), these methods can further reduce computational costs. Note that blocked cells that have neighbors that are not blocked cannot be culled and need to be processed in order to yield the proper collision effects.

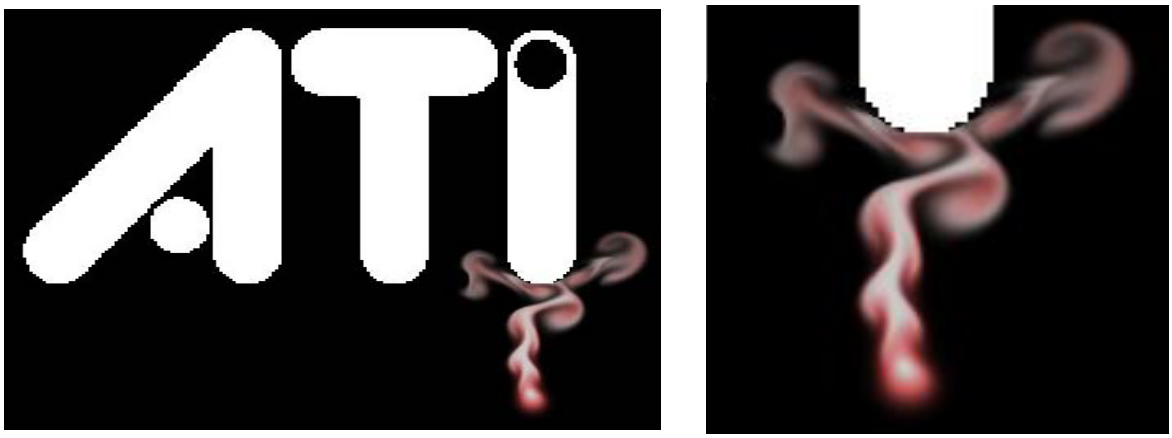


Figure 2 – Fluid flow with blockers

Early-Z and dynamic flow control: Multiple shadow maps with selective antialiasing

In this section, we describe a shadow algorithm that uses both early-z and dynamic flow control in order to cull computation in different steps of the algorithm. Shadow mapping's popularity as a shadow computation algorithm is due to the fact that it is an extremely fast method that maps nicely to graphics hardware. The drawback of shadow mapping is that it requires a very high resolution shadow buffer in order to properly sample the surface and reduce aliasing. Recently, several techniques have been introduced to mitigate the aliasing problem.

We present an approach that creates a series of "camera-chasing" shadow maps directly in front of the camera for nearby geometry, as well as a shadow map that encapsulates the entire scene geometry for the remaining geometry in the scene. Figure 3 shows a cross-section of two successive shadow map frusta along with the spherical regions that they encapsulate. For a given radius *r* for the sphere that is closest to the camera, the successive spheres can be computed using basic trigonometry. All shadow maps have the same resolution (e.g., 1024x1024), but note that the closer shadow maps have the required higher surface sampling rate, due to their tighter frusta.

The use of camera-chasing shadow maps significantly reduces aliasing, as evidenced by the higher sampling rate near the camera on Figure 4a. However, the hard "staircasing" artifacts are still clearly noticeable. In order to address this problem, we attempted several methods, such as bilinear PCF filtering and other fixed sampling kernels. We have found good results using the following two sequential steps:

1) Fetching from the shadow map using a disc of multiple sample offsets locally rotated by a per-pixel angle specified in an image-space lookup table.

2) Locally blurring the resulting frame buffer at the shadow transition regions.

The first step uses dynamic flow control to reduce the number of required texture fetches, while the second step uses Early-Z culling to avoid blurring regions of the screen that are not shadow transitions.

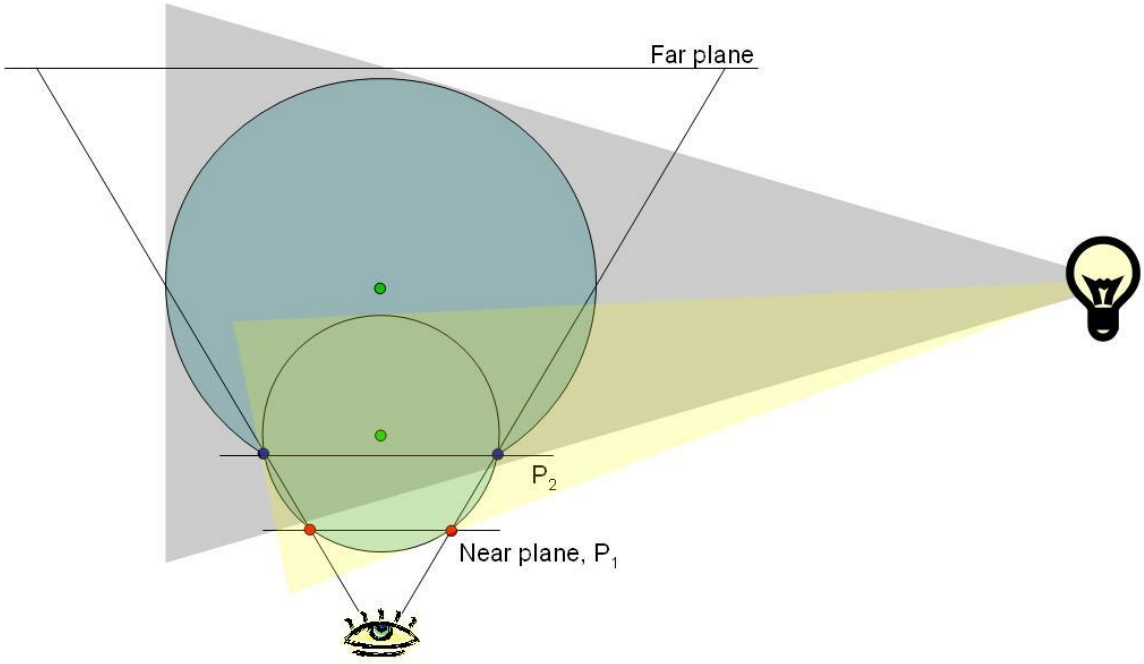


Figure 3 – Camera-chasing shadow maps

(a) Single fetch and no blurring (b) Multiple fetch and blurring

Figure 4 – Results with and without applying the rotated kernel lookup and the screen-space blur.

Shadow map sampling

In order to reduce aliasing artifacts, a common approach is to perform multiple shadow map lookups and apply percentage-closest filtering (PCF) [Reeves87]. In order to further reduce aliasing, we instead use a disc of 12 selected sample offsets and locally rotate these offsets using a per-pixel angle specified by an image-space lookup table.

Prior to performing the 12 shadow map lookups, we must determine which shadow map we want to use. We always use the closest one to the camera, whose frustum encapsulates the pixel (i.e., the one with tightest frustum). If we are using two shadow maps, we first project the sample using the tightest one, and if it is within range, we perform the 12 lookups using this shadow map, otherwise, we use the other shadow map, which encapsulates the entire geometry. Figure 4a shows the use of two shadow maps and a clear transition between the two.

Note that since this is a decision of whether to execute one code path or a different code path when shading the model, we cannot afford to do any early-z optimization, since that would require rendering the entire model multiple times. Thus, this is a clear case in which we want to perform the conditional inside the shader itself.

We have found that this approach, while further reducing aliasing artifacts, introduces some fine-grained noise. The shadow map blurring step described below addresses this noise.

Shadow map blurring

In order to address the fine-grained noise introduced by the above step, we perform a screen-space Gaussian blur pass on shadow transition regions by rendering a full-screen quadrilateral. Our image-space blurring technique is most similar to the work of [Arvo04], but is used to hide aliasing artifacts as opposed to computing penumbral opacity. The blur computation is only performed in the regions in which the shadow transitions from light to dark.

In order to efficiently blur those selected regions, when we render the geometry and perform the shadow map lookups, if the shadow tests differ within a given pixel, we store 1 in the alpha channel, otherwise we store 0. After rendering the geometry, in an extremely fast rendering pass, we transfer the alpha value to the z-buffer. Then, we perform the blur pass by setting the z test equal to 1. This only blurs the samples on shadow transitions, and avoids the expensive computation elsewhere.

Note that since the blurring computation is performed in image space, and the decision is whether to blur or completely cull the computation in that pixel, using Early-Z is far more attractive than using dynamic flow control. This final blurring step used a 5x5 Gaussian kernel at shadow transition regions and slowed down the algorithm by just 3% in our tested scenes. As shown in Figure 4b, this method results in no noticeable aliasing.

Dynamic flow control:
Shadow map space image processing for computation masking

There are other ways in which computation culling can be beneficial for shadow mapping. This section explains two methods for edge filtering and propagating the edge and depth information to encompass the boundary region that requires more processing. The first generates a mipchain for the shadow map edge mask, and simply fetches from a lower miplevel to effectively dilate the edge mask. The second propagates the min and max of the depth extent of the shadow map receiver texels to generate subsequent levels of a depth extent mipchain. This approach is particularly beneficial for complex scenes with a high depth complexity as it only applies the complex processing within a particular depth range per texel, and thus only the shadow map depths where the shadow boundaries appear in the final scene.

Conditional Processing for Shadow Mapping

A common artifact with shadow mapping is the resulting aliasing artifacts on the shadow map boundaries. The standard solution to avoid shadow map aliasing is PCF filtering, which usually requires taking several samples from the shadow map. While using high quality PCF filtering everywhere in the rendered scene provides correct results, it is not efficient. High quality filtering is only required for the regions of the scene where the shadow edges lie.

One simple and effective method for doing this is to only perform the high quality filtering in regions front facing to the light. If a projective spotlight texture or other form of gobo is used to light the scene, the shadow mapping computation should only be performed in the regions that the filtered light projects onto.

Incorporating this results in a huge speedup for the majority of cases, especially when the camera is facing the light, and most of the visible surfaces in the scene are backfacing with respect the light.

Computation Masking and Edge Dilation using Mipchain Generation

Another observation is that high quality PCF filtering is really only required near shadow boundary regions. In our implementation of PCF for anti-aliasing, the filtering kernel has a fixed width in shadow map space, so only regions within a certain distance from shadow boundaries in shadow map space need to be processed using the high quality PCF kernel. Regions completely inside or outside the shadow require only a single fetch from the shadow map in order to determine whether they are inside or outside the shadow. In order to determine the amount of filtering required per-pixel, we will build a computation mask to mask off regions that require higher quality filtering. There are a couple of approaches we use to determine these regions. We will describe them next.

The first technique involves edge-filtering the shadow map and propagating the filtered regions outward. Figure 5 shows an example of an edge mask generated from a shadow mask. A standard technique for propagating edge information outward is to perform some form of dilation operator multiple times on the image. However, doing this is expensive since the operator would have process all the pixels texels in the edge texture multiple times. However, since we are using this edge information for computation masking the dilation operation does not have to be exact. We do not need to know exactly which texels are within N texel lengths from an edge. As long as our computation mask encompasses the regions that require higher quality filtering, the results will still be correct.

A simpler alternative that is optimized for graphics hardware is to build a mipchain from the edge mask. By going down levels in the resulting mipchain, we have a lower resolution representation of the edge mask, where each texel occupies four times the area of texels in the previous level.

This effectively approximates the dilation of the shadow mask. Each time the image is filtered to go down a miplevel, the texel is dilated to twice its original size. So a dilation of 8 pixels in size can be achieved by going down log2(8), or three miplevels. However, there are a few implementation details that need to be considered.

The first is that the original edge filter for the highest resolution miplevel should be first thresholded so that edge pixels have a value of 1 and non-edge pixels have a value of zero. Then, when generating the subsequent miplevels, the filtering will average edge and non-edge pixels together, which will result in values between zero and one. To account for this, all non-zero pixels in the filtered edge mask should be considered to be in the dilated region.

The second consideration is that with standard fast 2x2 box filtering for mipchain generation, texel values only take into account texels from subsequent levels that project to within its footprint. For example, suppose we have a 16x16 texture which is completely black except for a single bright pixel at location (7, 7). (e.g. as close to the center as possible in the upper left quadrant) When generating the mipchain, intensity from the single bright texel is not propagated to the texels to the other 3 quadrants until the 1x1 mip level. Figure 6 visually shows an example of this.

In order to allow for proper propagation of intensity to implement dilation, when fetching from the miplevel, bilinear filtering should be used. Using bilinear filtering gives an extra texel width of dilation, with the texel width determined by the miplevel. For instance, a single texel from the mipmap three miplevels down is eight texels from the top-most miplevel in width. Filtering for mipchain construction is only performed up to the miplevel used as the computation mask.

During the scene rendering pass in the pixel shader the correct level of the mipchain is fetched from tex2dLOD function. If the result of the fetch is non-zero, more complex PCF filtering is performed, otherwise standard single sample shadow mapping is

used. Figure 7 shows some examples of the different criteria we have described so far for computation culling.

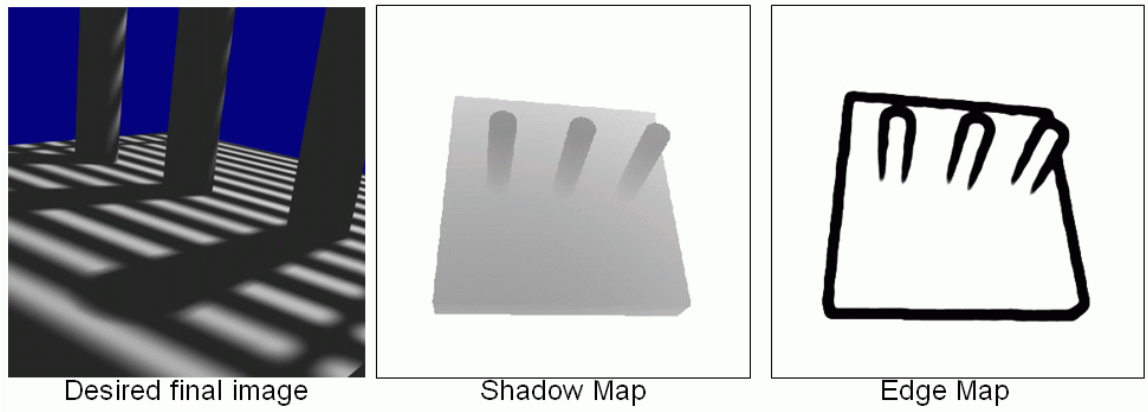


Figure 5 – The desired final image with soft shadows, the shadow map, and the edge map used for computation masking

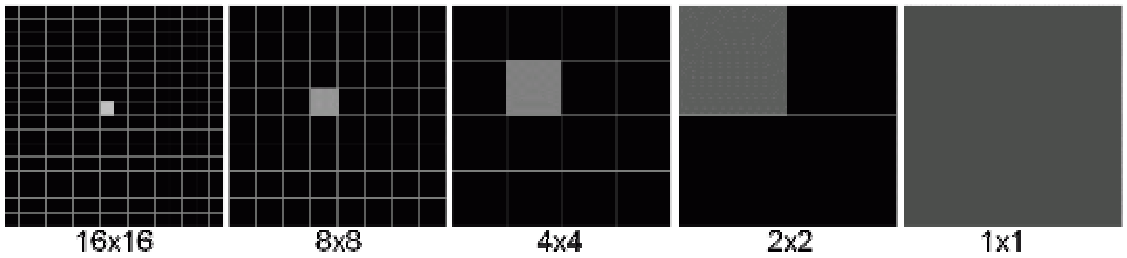


Figure 6 – Example of how mipchain generation propagates intensity values. As you can see, the single bright pixel does not get propagated to the other quadrants until the 1x1 level.

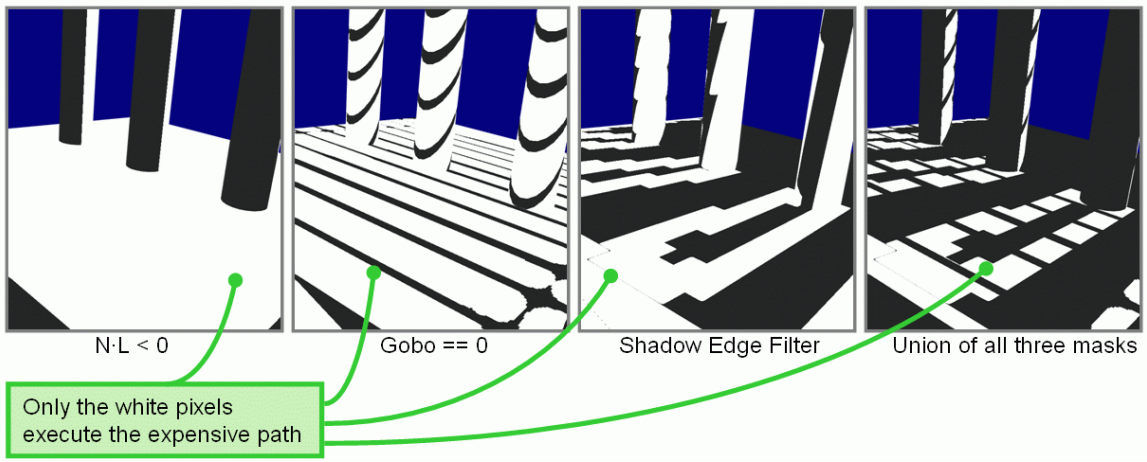


Figure 7 – Three different criteria used for computation masking for shadow mapping using PCF.

Depth-Extent Propagation for Computation Masking

Although using the dilated shadow map edge mask to limit the number of pixels that use high quality PCF filtering, there is still room for improvement. One issue is that the computation masking is based on the position of the projection of the current pixel into the computation mask. Any given texel in the computation mask can be though of as cutting a narrow light frustum through the scene within which all texels either perform high quality PCF or basic one sample shadow mapping. In the case where this texel frustum intersects many different objects in the scene, the complex shadow mapping applied to all pixels rendered within that frustum. Since this frustum emanates from the light source, the shadow boundary should only lie on the first or second object from the light source. In order to try to limit the high quality PCF processing to be performed only on the layer where it is required, we replace the dilated edge mask with a min and max distance from the light where the shadow can exist. We call this two channel texture the *depth extent map*.

To generate the depth extent map, the shadow map is edge filtered. If an edge is detected the value on the edge that is closest to the light is determined to be the blocker distance. Within the 3x3 neighborhood of the texel, the min and max of any shadow map depth values greater than the blocker value are stored. These can be considered to be the local range of depths for the receiver, (the surface the shadow is cast onto), and are written out to the depth extent map. The receiver depths are used, since the shadows lie on the receiving surface rather than the blocking surface. If an edge is not detected in the pixel shader, a degenerate depth range (min = 1.0, and max = 0.0) is written into the depth extent map.

In order to propagate the depth extent outward in texture space to generate the computation mask, we use a similar mipchain technique as before. However, in this case, lower miplevels will be used to store the overall depth extent of all the depth extent texels in a small neighborhood. Because of this we need to compute the mipchain using our own pixel shaders.

Since we are propagating neighborhood min and max values, using box filtering or bilinear will not work correctly. Our solution is to compute the region min and max of the depth extents of a 3x3 neighborhood in the previous mip-level. The 3x3 neighborhood is used rather than a 2x2 neighborhood since bilinear filtering can not be used to propagate depth extent information. The 3x3 neighborhood extends one extra pixel to the right and downward in order to pull depth extent information from other quadrants and subquadrants. Since we use min and max operations over the neighborhood, non-edge pixels (which have degenerate depth extents having a max of 0 and a min of 1) do not affect the depth extent. If all texels in the 3x3 neighborhood are non-edge, it naturally falls out of the math that the resulting texel in the next level is non-edge.

Just as in the previous section, the dilation amount should be determined by the width of the PCF kernel in shadow map space. The number of the miplevel chosen in the

min-max depth extent mipchain is equal to the log2 of the kernel width. Neighborhood min-max mipchain generation only proceeds until this miplevel.

The depth extent is fetched using the same texture coordinates used to fetch from the shadow map. When rendering the scene pass, the high quality PCF filtering is only performed if the current depth value is within the depth extent fetched from the depth extent mipchain. If not, basic single sample shadow mapping is used.

In most cases we have seen an overall performance increase (up to 3x speedup) from using these approaches in our test scenes. Generally adding conditionally based computation masking to a shader will help the average case at the expense of the worse case performance due to the additional overhead of the conditional instructions.

Any performance increases are dependent on the scene and viewing angle. A complex scene consisting mostly of shadow edges may require high quality filtering over nearly the entire image, and might not see an overall improvement in performance.

Conclusion

In order to efficiently render a scene, one important consideration is to determine how to best take advantage of features of current graphics hardware that provide the ability of efficient computation culling. In these notes we outlined the benefits and costs of two different optimizations, and we showed examples of applications in which computation can be completely avoided for certain pixels via early-z culling, and applications in which it is significantly beneficial to avoid specific code paths via dynamic flow control.

References

[Arvo04] Arvo J., Hirvikorpi M., and Tyystyjarvi, J. 2004. “Approximate Soft Shadows Using an Image-Space Flood-Fill Algorithm”. Computer Graphics Forum (EUROGRAPHICS’2004),

[Reeves87] Reeves, W. T., Salesin, D. H.., and Cook, R. L. 1987. “Rendering Antialiased Shadows with Depth Maps”. SIGGRAPH, 1987, pp. 283-291.

[Sander04] Sander, P. V., Tatarchuk, N. and Mitchell, J., "Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering,” ATI Technical Report, August 2004.