PCU: The Programmable Culling Unit

Jon Hasselgren Tomas Akenine-Möller

Lund University

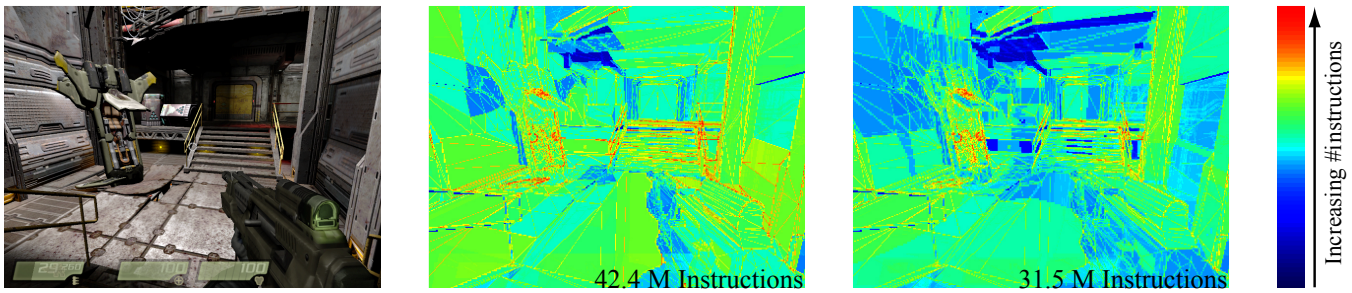


Figure 1: *Our programmable culling unit applied to Quake 4 (courtesy of id Software). Left: Rendered image. Middle: False color image of the number of instructions executed per pixel using a standard GPU. Right: Similar visualization using our programmable culling unit.*

Abstract

Culling techniques have always been a central part of computer graphics, but graphics hardware still lack efficient and flexible support for culling. To improve the situation, we introduce the programmable culling unit, which is as flexible as the fragment program unit and capable of quickly culling entire blocks of fragments. Furthermore, it is very easy for the developer to use the PCU as culling programs can be automatically derived from fragment programs containing a discard instruction. Our PCU can be integrated into an existing fragment program unit with a modest hardware overhead of only about 10%. Using the PCU, we have observed shader speedups between 1.4 and 2.1 for relevant scenes.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

Keywords: rasterization, shaders, hardware, culling

1 Introduction

Faster rendering is a major field in computer graphics research due to the ever-increasing demands of applications. A prime example of such applications are games with high image quality, complex shading, and detailed geometry. In real-time graphics, programmable vertex, geometry, and fragment shaders provide the programmers with more flexibility for the rendering tasks, but higher image quality and using more complex shading incur a cost in performance.

The fragment pipeline is often considered to be the major bottleneck in a GPU [Aila et al. 2003], and hence it is a clear candidate for acceleration algorithms. At a high level, the task of the fragment pipeline is to *access memory* and *execute instructions* in order to produce the output (e.g., color, depth, stencil) of a fragment. The focus of our research is on the latter.

To make shader programs run faster, one can insert `KIL` (fragment discard) instructions at appropriate places, in order to provide an *early-out*. In practice, these instructions seldom make the execution faster on contemporary GPUs, due to the underlying hardware design. In addition, a GPU designer can employ pipelining and parallelization techniques for faster shader instruction execution. Note, however, that the fastest instruction is the one that never is executed to begin with. Hence, the target of our paper is to avoid executing, i.e., *cull*, a substantial amount of instructions which do not contribute to the final image.

To that end, we present the *programmable culling unit* (PCU). The idea is to execute a *cull program* on tiles of pixels before the per-pixel shader processing starts. In many cases, a conservative decision can be made based on the outcome of the cull program, and per-pixel executions completely avoided. For example, if all pixels in a tile are in shadow, it may be possible to determine this on a tile-level, and shading instructions can be avoided for all pixels in the tile. Our PCU is flexible in that the programs can be automatically generated from the fragment shader programs (i.e., transparent to the programmer), or written explicitly. The core idea of the PCU is to use *interval arithmetic* [Moore 1966] to bound the value of a floating-point variable, and let the fragment pipeline operate on such bounded number representations.

The PCU makes existing fragment programs run faster, and opens up new possibilities for culling algorithms specifically written for the PCU. In addition, a major advantage is that the rendering programmer can work in a more carefree way. For example, many current games are doing back-face culling from the light source on the CPU, before sending the geometry to the GPU for shading. Such culling can be done automatically on a per-tile basis using the PCU. For an example of how our PCU can improve performance, even in a highly optimized computer game, see Figure 1.

A prototype hardware implementation of a fragment pipeline has been developed, and the increase in size is about 10%. We provide extensive simulation results using a variety of benchmarks, such as commercial games and relevant shader effect demos. For these, we observed that 29–52% of the instructions can be avoided, which implies a speedup of $1.4 - 2.1\times$. The memory bandwidth utilization is also decreased by approximately 15%.

ACM Reference Format
Hasselgren, J., Akenine-Möller, T. 2007. PCU: The Programmable Culling Unit. *ACM Trans. Graph. 26*, 3, Article 92 (July 2007), 10 pages. DOI = 10.1145/1239451.1239543 http://doi.acm.org/10.1145/1239451.1239543.

2 Previous Work

In our research, we have used *interval arithmetic* [Moore 1966] (IA), which is an arithmetic defined on intervals rather than real numbers. It dates back to the early 20th century, and is now an established field of research. IA has already been used on several occasions in the field of computer graphics, and we will mention only the ones that relates to our research. For a more in-depth overview, consult Kearfott's survey [1996].

The Achilles heel of interval arithmetic is that the width of an interval can grow rapidly for complex equations. Affine arithmetic [Comba and Stolfi 1993] reduces this problem by tracking all linear dependencies in an equation and evaluates them exactly. Higher order dependencies are solved using linear approximation and by adding an extra term that bounds the approximation error.

Culling To speed up execution in both software and hardware, it is common to use culling algorithms. Hardware culling is typically limited to fixed function mechanisms, such as back-face culling or hierarchical depth culling [Greene et al. 1993; Morein 2000].

Hierarchical depth culling (HDC) is an optimization of depth buffering. When a tile of fragments is about to be rasterized, a conservative test is performed to prove whether that tile is covered by the contents in the depth buffer. If it is, the entire tile can be skipped which results in performance gains. Aila et al. [2003] improve the culling rate of HDC by introducing a delay stream, which basically performs depth buffering first, and then puts the triangles on hold until the "occluding power" of the depth buffer has been built up. Another variant of HDC, called Z-min culling [Akenine-Möller and Ström 2003], avoids depth reads when a tile of fragments being generated are in front of the contents of the depth buffer.

In an attempt to implement more flexible hardware culling, Purcell et al. [2003] introduced *computation masks*. The basic idea is to let a fragment program set up the depth buffer so that all fragments that can be culled fail the depth test. The hierarchical depth culling hardware will then perform the actual culling. The weak points of computation masks are all related to overlapping geometry. First, the computation mask will overwrite the content of the depth buffer, which makes it hard to use depth testing in conjunction with computation masks. Second, the computation mask is given in screen space with just one entry per pixel. This makes it difficult to make different culling decisions for overlapping geometry.

Occlusion queries provide functionality to query the graphics hardware for the number of fragments that pass the depth test. The prime example of using occlusion queries is to draw the bounding box of a mesh. If no fragments pass the depth test, the entire mesh is occluded and does not need to be rendered. These algorithms require intervention by the programmer, and it can be difficult to achieve good performance due to the latency and overhead of a query, or require sophisticated CPU algorithms [Bittner et al. 2004]. In general, occlusion queries are orthogonal to our work.

Programmable Shaders The core feature of modern GPUs is the availability of programmable shaders. These are commonly divided into geometry shaders [Blythe 2006], vertex shaders [Lindholm et al. 2001], and pixel/fragment shaders. However, as the programming languages are very similar, the implementation trend is to use unified shader units [Doggett 2005], which can execute the program for any type of shader. This enables more efficient load balancing between the different tasks. For instance, if a scene contains many small triangles, more processing units can be allocated for vertex processing. On the other hand, if we draw a typical GPGPU full-screen quad, all units can be allocated for fragment processing.

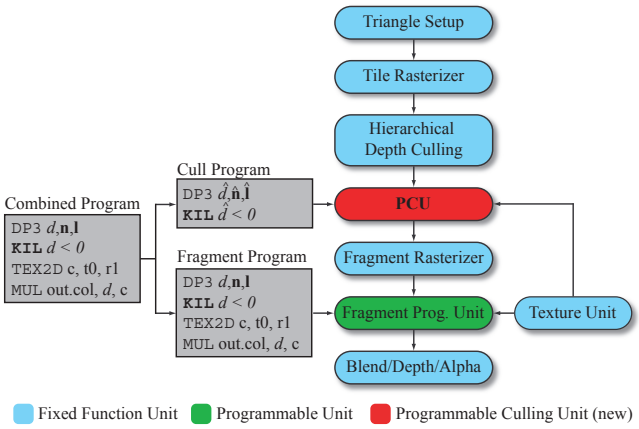


Figure 2: *Behavioral model of our proposed rasterization architecture. We add a new programmable unit between the depth culling unit and the fragment-producing unit. Our unit executes a cull program, and decides whether a tile can be culled or not based on the results of the program.*

There has been some work made on software evaluation of programmable shaders using interval or affine arithmetic. Greene and Kass [1994] evaluated shade trees [Cook 1984] using interval arithmetic. This was used to compute bounds for the final color over a surface area element. The width of the bounds was used to guide adaptive refinement antialiasing. Heidrich et al. [1998] used affine arithmetic to evaluate procedural shaders written in RenderMan. Doing so enabled them to sample a procedural shader over an area element. The authors argue that this can be used to, for instance, adaptively sample procedural textures, or taking material into account in bounded lighting equations [Stamminger et al. 1997]. Moule and McCool [2002] use interval arithmetic to bound the approximation errors when performing adaptive tesselation of displacement maps. As a hierarchical representation of the displacement map function, they use a mipmap structure of intervals.

3 Programmable Culling Unit Overview

In this section, we give a high-level overview of our programmable culling unit, and motivate some early design choices.

Figure 2 shows a behavioral overview of our novel rasterization architecture, which includes the PCU as a new unit. The PCU is the last unit which processes entire tiles of fragments. It conservatively decides whether to terminate the tile or send it down the pipeline for further per-fragment processing. The fundamental difference, compared to hierarchical depth culling (covered in Section 2), is that the PCU bases its decision on the output from a shader program execution, rather than fixed function computations.

One of our main ambitions was to make it as simple as possible for the programmer to take advantage of the PCU. With our approach, the programmer does not have to care about new programming languages, writing conservative algorithms, or taking tile sizes into account. The whole PCU can simply be seen as a very fast `KIL` (fragment discard) instruction that operates on a per-tile basis.

As an example, consider the "combined program" in Figure 2. This program performs diffuse lighting by computing the dot product between the normal and light vectors, and multiplies the result by

a diffuse material coefficient stored in a texture. In this case, the programmer added a KIL instruction to terminate fragments where the normal does not face the light. We see this KIL instruction as an opportunity for culling a whole tile of fragments. In order to do so, we need a way to conservatively prove that the condition for the KIL instruction is fulfilled for every fragment within the tile. From this follows that we must also be able to conservatively evaluate the DP3 instruction, since the KIL instruction depends on its result. We must also be able to find conservative bounds of the input (the normal and light vectors in this case) for a whole tile, since the DP3 instruction in turn depends on these values.

In order to implement this chain of conservative evaluations, we base the PCU on the same instruction set as the fragment program unit. However, instead of floating-point variables as source and destination registers to an instruction, we use *intervals* and implement the instruction using the principles of interval arithmetic [Moore 1966].[1] As a simple example, consider a standard ADD instruction:

$$\texttt{ADD}\ c,a,b \quad \Longleftrightarrow \quad c = a+b \tag{1}$$

For the corresponding PCU interval instruction, we replace the operands by intervals, $\hat{a},\hat{b},\hat{c}$, where an interval, e.g., $\hat{a}$ is defined as:

$$\hat{a} = [\underline{a},\overline{a}] = \{x \,|\, \underline{a} \le x \le \overline{a}\}. \tag{2}$$

The PCU interval ADD instruction is then:

$$\texttt{ADD}\ \hat{c},\hat{a},\hat{b} \quad \Longleftrightarrow \quad \hat{c} = \hat{a}+\hat{b}, \tag{3}$$

where the interval addition operation is implemented as:

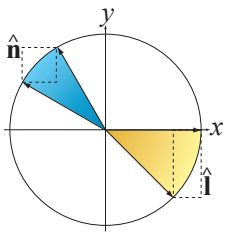
$$\hat{a}+\hat{b} = [\underline{a},\overline{a}]+[\underline{b},\overline{b}] = [\underline{a}+\underline{b},\overline{a}+\overline{b}]. \tag{4}$$

As can be seen, the result of the interval addition contains all possible results of "normal" additions, or more formally, it holds that $a+b \in \hat{a}+\hat{b}$ given that $a \in \hat{a}$ and $b \in \hat{b}$. It is therefore conservatively correct. In similar fashion, we redefine the behavior of every instruction in the fragment program instruction set. Full details of our interval instructions, including conservative texture lookups, can be found in Appendix A.

In addition to using interval instructions, the input must also be defined as intervals. Therefore, we must be able to compute conservative bounds for quantities interpolated over an entire tile of fragments. This is treated in more detail in Section 4.2.1.

Given this architecture, we can execute the shader program for an entire tile at a time. When a KIL instruction is executed, we determine if the conditional expression is unambiguously fulfilled by checking the interval of the input. In such a case, we can quickly discard the entire tile and its fragments, which saves valuable processing time. Otherwise, per-fragment shader execution follows.

2D Interval Dot Product Example Let us once again revisit the example in Figure 2, while also looking at the figure to the right. For an entire tile of fragments, assume that we have determined that the input interval of the normals is $\hat{\mathbf{n}} = ([-\sqrt{3}/2,-1/2],[1/2,\sqrt{3}/2])$, and the interval for the light vector is $\hat{\mathbf{l}} = ([1/\sqrt{2},1],[-1/\sqrt{2},0])$, as illustrated in the figure. The dot product between these interval representations results in $\hat{d} = \hat{\mathbf{n}} \cdot \hat{\mathbf{l}} = [-(\sqrt{6}+\sqrt{3})/\sqrt{8},-1/\sqrt{8}]$ (see the DP3 instruction in Table 2). We then reach the KIL instruction and note that the operand, $\hat{d}$, can be at most $\overline{d} = -1/\sqrt{8}$. Since this value is strictly less than zero, we can cull this whole tile without executing the fragment program for every fragment. This is the source of the performance gain in our algorithm. □



[1] We decided to use interval arithmetic because it is simple to implement in hardware. An alternative approach is to use affine arithmetic [Comba and Stolfi 1993]. See Section 7 for further discussion.

4 PCU Interaction

In this section, we will treat the PCU as a black box which takes intervals as input and determines whether a tile can be culled. It is important to note that the PCU is a part of a much larger system including both hardware and the driver. In this section, we focus on the interaction between that system and the PCU.

4.1 Driver

Here, we will introduce a new CUL instruction, present compilation issues, and discuss optional instruction support. All these features are implemented in the driver.

4.1.1 CUL Instruction

The system in Section 3 allows for rapid culling of fragment programs containing KIL instructions, but we also introduce a CUL instruction that can improve performance in some cases. As an example of when it is useful, consider a fragment program designed for multi-pass shading, such as the program from Figure 2. These programs rarely or never use a KIL instruction to discard fragments whose normals face away from the light. The light contribution will be zero anyway, so the KIL instruction is not needed for correctness and will in most cases only reduce performance.

In this case, we want the KIL instruction to exist only in the cull program, which is exactly how we define the CUL instruction. The CUL instruction, and all instructions depending on it, will only be propagated to the cull program. Thus, where a KIL instruction is always guaranteed to discard a fragment if the condition is fulfilled, a CUL instruction may or may not discard the fragment depending on if it belongs to a tile that can be culled. The CUL instructions can be removed for hardware not supporting cull programs.

4.1.2 Program Compilation and Separation

A key point of our programmable culling unit is the ease of use. The programmer writes a combined program, and it is up to the driver or compiler to separate the cull and fragment program. This process can be done easily and efficiently using dead code elimination [Cytron et al. 1991]. For the fragment program, we mark the color outputs, depth outputs, and all KIL statements as live code. We then perform dead code elimination, efficiently removing all code not contributing to the final result. For the cull program, we mark all CUL and KIL statements as live, and again perform dead code elimination to remove all code not contributing to the result.

It should be noted that dead code elimination is a lightweight optimization technique that does not introduce any significant performance impact on the driver, even if it is carried out at runtime.

4.1.3 Optional Instruction Support

A convenient feature of our PCU, and culling algorithms in general, is that lack of culling does not compromise the correctness of the result. An implementation may therefore freely omit PCU support for any instructions or features present in the fragment program unit. The driver can then examine all KIL and CUL instructions and determine if they depend on some unsupported feature. If this is

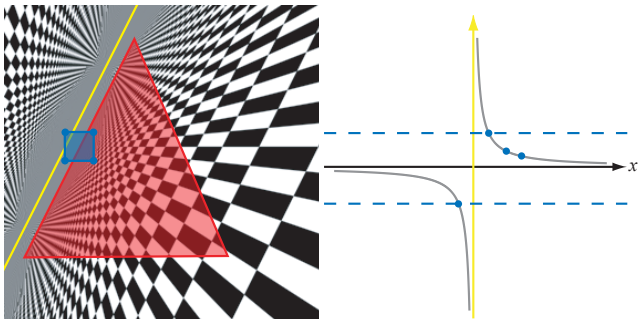


Figure 3: *The perspective-correct interpolation problem. The left figure shows the perspective-correct interpolation space of a triangle, when extended outside the triangle boundaries. Note the yellow line where the checkerboard is mirrored. This is where the horizon of the triangle would be projected if it was infinitely large. The right figure shows the perspective-correct interpolation function as gray curve, and the projection line (at $x = 0$) marked in yellow. Note that computing bounds using the corners of the tile (in blue) gives an incorrect result.*

the case, the `KIL` or `CUL` instruction is simply marked as dead code when extracting the cull program.

We use this strategy to handle *dynamic* loops (loops with a variable number of iterations). For simple *static* loops, with a constant number of iterations, we can simply use loop unrolling. However, dynamic loops present more of a challenge, and we do not currently support them in the cull program. There are ways of reformulating a dynamic loop on interval form [Heidrich et al. 1998], but the problem is that we cannot guarantee that a loop converging on a per-fragment level will also converge when using our interval instructions. There is a possibility of undesired infinite loops.

To support dynamic loops, we believe the best approach would be to extend the instruction set of the PCU with instructions for this purpose, and to force the programmer to write a custom cull program. Thereby, the programmer gets the responsibility to formulate a loops that converge. This is left for future work.

4.2 Hardware

When considering the PCU as a black box, there are still some issues that can be discussed when it comes to hardware. In this subsection, we will first discuss how to compute the input intervals to a tile, and then present a solution to how small triangles can be handled efficiently.

4.2.1 Interpolation

Given an implementation of the instruction set in Appendix A, we may execute a fragment program for a whole tile of fragments. However, in order to do so, we also need to compute bounding intervals for the varying (or interpolated) inputs.

We use a method inspired by the depth bounds computations of hierarchical depth culling. Initially, we compute the value of the varying attribute in all four corners of the tile using interpolation. We then compute the bounding interval of these four values, and call it $\hat{a}_c = [\underline{a_c}, \overline{a_c}]$. We also compute the bounding interval of the varying attribute at the triangle vertices, and call it $\hat{a}_{tri} = [\underline{a_{tri}}, \overline{a_{tri}}]$. The final bounding interval of the varying attribute over the tile can be computed as $\hat{a}_{tile} = [\max(\underline{a_{tri}}, \underline{a_c}), \min(\overline{a_{tri}}, \overline{a_c})]$.

Finally, we must take care of a special case, which is illustrated in Figure 3. Here, perspective-correct interpolation over a triangle is illustrated in form of a checkerboard texture. As can be seen, the texture is mirrored about the yellow *projection line*, which is where the horizon of the triangle would project if it was infinitely large. This mirroring effect is a form of back-projection caused by the division used in perspective-correct interpolation.

Now, assume we wish to compute the bounding interval of some varying attribute over the blue tile, which overlaps the projection line. The right part of Figure 3 shows the perspective-correct interpolation function, as well as the values we get when we interpolate the four corners of the tile. Note that the bounding interval of these corners (dashed blue lines) is obviously incorrect since the function approaches infinity at the projection line.

We handle this special case by setting $\hat{a}_{tile} := \hat{a}_{tri}$ as the bounding interval for tiles overlapping the projection line. One might argue that this interval is overly conservative, but these problematic tiles are so rare that it is hard to motivate more complex computations. In our implementation, we only traverse tiles actually overlapping the triangle, and use perspective-correct barycentric coordinates [McCool et al. 2002] to do the interpolation. We can easily detect the problematic tiles when computing perspective-correct barycentric coordinates for the corners of a tile. The perspective-correct barycentric coordinates are expressed as a rational function, and if the denominator is less than zero for any of the tile corners, then the tile crosses the projection line.

4.2.2 Higher Level Culling

A potential weakness of our PCU is that the culling is done on a per-tile and per-triangle basis. In the case of micro-polygons, triangles can be smaller than a pixel, which means that executing the cull program will cost at least as much as executing the fragment program. Here, we describe a solution to this problem.

In order to handle micro-triangles efficiently, we use a delay stream like unit [Aila et al. 2003]. This unit receives triangles in the same order as they are sent to the graphics card. It also keeps an internal state with bounding intervals of the varying attributes. The unit groups triangles as long as the accumulated bounding interval for the *position* attribute is smaller than the size of a tile. This means that grouping is done as long as the bounding box of the triangles is smaller than the size of a tile. When the accumulated triangles reach this limit, we execute the cull program using the accumulated bounding intervals. If the outcome indicates that culling can be done, the entire set of small triangles is terminated. Otherwise, per-fragment processing commences for each triangle in the set.

Triangles overlapping more than one tile are rendered as usual. That is, we execute the cull program for every tile during rasterization.

4.2.3 Fragment Program Switching

In some cases, such as shader level of detail, it may be convenient to be able to use the cull program to specify which, out of many, fragment program should be executed for the fragments in a tile. A nice example of this is our “Soft Shadows” scene in Section 6. Here, we may skip a considerable amount of shadow map lookups if we can prove that a tile is entirely lit, but we must still perform the lighting on a per-fragment level so we cannot completely cull the tile.

We can support this if the underlying hardware is capable of rapidly switching between a small number of fragment programs. In this case, we simply attach a fragment program index to each cull instruction, and use the corresponding fragment program when rendering the fragments of a tile that has been culled by that instruction.

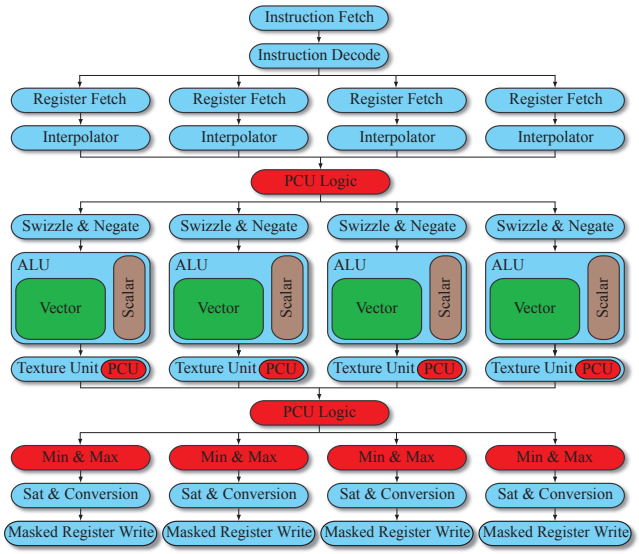


Figure 4: *Conceptual diagram of a contemporary fragment program unit which we have extended to a PCU. The new units are marked in red.*

5 Implementation - Combined Shader Unit

So far, our description of the programmable culling unit has been on a functional level. However, our goal has been to reuse existing hardware when possible, so it is only natural to combine the PCU and the fragment program unit (or the unified shader unit, if such an architecture is used) into a single programmable unit.

Figure 4 shows a block diagram of a possible fragment program unit. The execution pipelines are organized as four separate parallel units, with a direct mapping to a quad of 2×2 fragments. This enables simple approximations of derivatives by finite differences, which is essential for correct mipmapping. The fragment program pipeline is broken down into several steps, and in order to avoid hazards, it is reasonable to assume that the GPU executes at least as many parallel program threads as there are pipeline steps in the program unit.

In order to implement our PCU, we augment the fragment program unit with new logic shown in red in Figure 4. These additions mainly consists of extra control logic before and after the ALU. The logic is responsible for value rerouting, detecting and handling the special cases of Table 2, and handling the special case for interpolation. There are also additions in the texture unit to handle mipmap computations and filtering, and a final set of min & max units to assemble the result. These units may be taken from the ALU and moved to the end of the pipeline. No extra hardware is needed in this case.

We have implemented a simple fragment program unit augmented with a subset of the PCU, in VHDL. We restricted our hardware implementation to the arithmetic and logic functions, while omitting texturing and interpolation. The reason for this was that we had no access to peripheral units, such as texture caching, vertex buffers, and so forth. Admittedly, our implementation is only a fraction of a GPU, but implementing the entire hardware would take considerable effort. In addition, for the same reasons, our processor model is most likely much simpler than what can be found in the GPUs on the market. However, we learned two facts which we believe will generalize to any GPU design. First, we can greatly reuse the computational resources and only need to add a small amount of logic to handle rerouting and special cases for interval arithmetic. In our particular implementation this overhead amounts to about 8%, and in this case we used duplicated min and max units. Second, using interval arithmetic increases the latency for executing an instruction. We therefore need more executing threads in order to avoid hazards, and consequently need a larger register file in on-chip memory. In our particular implementation, the execution pipeline increased from 22 to 24 steps, which gives a 9% increase in register space.

Note that we implemented the PCU by extending a fragment program unit. This indicates that it should be relatively straightforward to make the PCU a part of a unified shader architecture. No matter what architecture we choose, it should be noted that the PCU introduces an additional delay in the pipeline, since we add an additional program that needs to be executed. However, we do not believe this will have any negative effect on the throughput.

6 Results

We have evaluated our PCU experimentally using a variety of modern shader based applications, all rendered at a resolution of 1280×1024. The evaluations are based on a functional simulator written in C++, which implements all features of the PCU specified in this paper, as well as most other features of modern graphics hardware. This includes interval-based texture lookups, texture caching, HDC, and depth & color buffer compression and caching. We implemented the PCU on top of a normal rasterizer, and also implemented an "optimal" culling unit for reference. In practice, we implement this optimal unit by executing the cull program for every fragment overlapping the triangle inside a tile, but without counting the instructions. We then make the culling decision based on the results of all fragments. This result can be seen as an upper bound for culling when using a particular program and tile size.

Quake 4 is a modern game based on the Doom 3 engine which uses advanced per-pixel lighting, bump mapping, and stencil shadows. We used a logging OpenGL driver to output frames from the actual game, which means that we can completely recreate its behavior.

The only modifications we made to the logged data was to enhance the fragment programs with culling code. Since we did not have access to the game code or any geometric data, we chose to disable shadowing in the game and concentrated our optimizations on the lighting only. With access to the full code it might be possible to use more sophisticated culling, such as clipping shadow volumes against portal frustums. We got the best balance of performance and culling rate when applying simple culling algorithms such as backface (bump-mapped), attenuation, and spot-light frustum culling.

It should be noted that the Doom 3 game engine is optimized to perform a fill rate heavy task on a wide range of graphics cards with varying performance. It therefore makes heavy use of CPU culling, and possibly even pre-computations such as splitting geometry by static lights in order to save fill rate. Hence, as it is already optimized, this can be considered as a very difficult case for our PCU.

Soft Shadows is based on the soft shadows demo in the NVIDIA OpenGL SDK [Uralsky 2005], which we replicated using our logging driver. This program uses a shader that draws eight jittered samples from the shadow map. If the shadow status for the samples varies, the fragment is assumed to be in the shadow penumbra and an additional 56 samples are drawn.

For this scene, we used a custom cull program. The reason is that we can use the powerful interval-based texture lookup (see Appendix A.2) to determine if a fragment is outside the penumbra. We do this using just one interval texture lookup, where the tex-

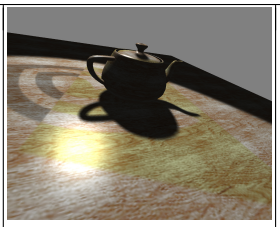
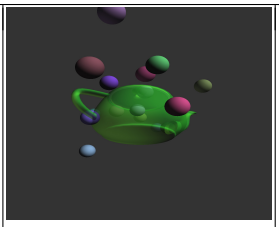
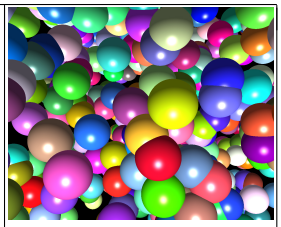
					
	Quake 4		**Soft Shadows**	**OIT**	**Spheres**
Scene	**A**	**B**			
Dynamic instructions	143 M	165 M	481 M	54 M	187 M
Instruction ratio PCU	68% (**1.5**×)	71% (**1.4**×)	48% (**2.1**×)	61% (**1.6**×)	49% (**2.0**×)
Instruction ratio optimal	60% (1.7×)	60% (1.7×)	N/A	56% (1.8×)	45% (2.2×)
Tile cull ratio PCU	40%	31%	57%	78%	58%
Tile cull ratio optimal	40%	37%	N/A	83%	82%
Total BW	**82%**	**85%**	**86%**	**87%**	**72%**

Table 1: *The top section shows performance figures in terms of dynamic instructions, i.e., the total number of instructions executed when rendering a frame. We first show the number of instructions used by a normal architecture. Thereafter follows the number of instructions executed with the PCU (in percent of the original, i.e, the lower the better), and the corresponding speedup. Note that this figure includes the PCU interval instructions, where each such instruction counts as four normal fragment program instructions. Finally, we show the same statistics for an "optimal" culling unit, as described in Section 6. This unit is assumed to execute the cull program in zero time, so the culling instructions are not counted. In the middle section, we show the cull ratio in percentage of the tiles that actually use a cull program. Note that the instruction ratios include all instructions, even those when a culling program cannot be used, which explains why the tile cull ratio is better than the instruction ratio. In the lower section, we present bandwidth figures for our PCU in percent of a standard architecture. We only present the total bandwidth since the gains were quite evenly distributed over texture, depth buffer, and color buffer bandwidth.*

ture coordinate interval is computed so that it covers the entire filter used for sampling. We can then determine if a tile is entirely in shadow or entirely lit. If the tile is in shadow, we immediately discard it, and if it is completely lit, we skip all shadow map sampling in the fragment program. Finally, if the tile is in the penumbra region we use the original fragment program. In addition, we have implemented back-face and attenuation culling for the light source. We do not present any results for the optimal culling unit for this example, since it is not completely defined how the interval texture lookup would work in the optimal case.

Order Independent Transparency (OIT) is based on the depth peeling algorithm [Mammen 1989]. Once again we used our logging driver and an example found in the NVIDIA OpenGL SDK. Depth peeling is an iterative multi-pass algorithm which creates one depth layer per pass. This is done by holding the depth values of the previous pass in a texture, and performing a two-sided depth test. This test discards all fragments with a depth component less than or equal to the value of the previous pass, or greater than the current value in the depth buffer.

Since the original shader code already contained a `KIL` instruction, we made no significant modifications to it. We simply noted that the PCU provides a two-sided hierarchical depth test, while a normal architecture only provides a one-sided hierarchical depth test (namely the depth buffer).

The results for this scene were generated under the assumption that a "sparse mipmap" of the depth buffer is available, as discussed in Appendix A.2. In this case, it simply means that the tile's Zmin and Zmax of the depth buffer are available in the PCU. This is perfectly reasonable since modern hardware supports Zmin/Zmax-culling. It should be noted that the number of instructions when using our PCU is reduced by an additional factor of two if a sparse mipmap is available for the color buffer as well. In this case, we get a massive speedup of a factor 3.7.

Spheres is a simple test scene that renders procedural spheres using billboards and a fragment program. A `KIL` instruction is used to discard all fragments in the billboard which do not overlap the sphere. The overlap test is performed in two-dimensional texture space.

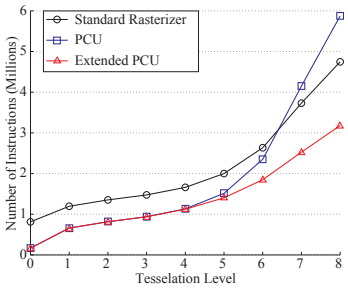
An important feature for this test scene is that we also compute a custom depth value in the shader, which is simply the depth of the sphere at each pixel. This typically breaks hierarchical depth culling functionality on current hardware. However, with our PCU it is a simple extension to output a bounding interval for the depth values in a tile, and we can then feed the bounding interval directly to the hierarchical depth culling unit. This way, we can efficiently handle depth culling of shaders with custom depth computations, which was previously an unsolved problem. This is desirable in many applications, such as rendering curved surfaces [Loop and Blinn 2006] and parallax mapping [Tatarchuk 2006]. It should be noted, however, that we must execute the cull program before the hierarchical depth test in this case. This opens a question whether the PCU should be placed before or after the hierarchical depth unit. Preferably, the order of the units should be configurable.

Discussion of Results The results of our evaluations are summarized in Table 1. Note that the PCU performance figures include execution of the culling program. As can be seen, we achieve significant performance improvements for all test scenes. The improvements are larger for the simpler demos than for the game scene, as expected, since the game already makes heavy use of CPU-based culling. Still, we believe that a performance improvement of 1.4× is significant considering how easy it was to modify the game. Notice also how well our culling unit performs when compared to the "optimal" case for that scene and program. In the bottom part of Table 1, we present bandwidth figures. Bandwidth reduction was not our primary goal, but even here we noticed improvements when compared to a standard GPU.

6.1 Small Triangles

We also examined how our PCU architecture scales with decreasing triangle size, with and without the extension presented in Section 4.2.2. In this test, we used a very simple scene in order to focus

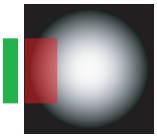
entirely on the effects of varying tessellation. The scene consists of a unit sphere that is clipped arithmetically by a cylinder in the fragment program. We varied the tessellation of the sphere in order to study how the algorithm performs at different triangle sizes.



We evaluated three different rasterization algorithms, and the results are shown in the diagram to the right. The standard rasterizer emulates a modern hardware rasterization architecture. That is, it is based on pixel quads as the smallest set of pixels that can be rasterized. It should be noted that it is therefore sub-optimal for rendering extremely small triangles, but this is a problem inherent in most current hardware rasterizers. The PCU algorithm refers to our PCU architecture without the extension in Section 4.2.2. As we predicted, the performance decreases more rapidly with increasing tessellation than for a standard rasterizer. In this particular scene, we have found that the two curves intersect when the average projected triangle area is around three pixels. Finally, we have the extended PCU algorithm as discussed in Section 4.2.2. Note that the extended PCU now follows the same trend as a normal rasterizer, with a virtually constant performance advantage due to the culling.

7 Discussion

The prime use of our PCU is to speed up fragment shader execution using a very simple culling program. We believe this is the reason why we have achieved such good results using interval arithmetic, which may otherwise suffer from rapidly growing bounds when computations become too complicated. This has terminated many efforts of implementing interval-arithmetic-capable CPUs. However, in culling it is more a rule than an exception to use a very simple algorithm, and this makes interval arithmetic well-suited for the task.



During the course of this project, we noted that existing fragment shader programs can be used by the PCU without any additional work by the programmer. However, with some additional effort, much better culling can be obtained in many cases. For instance, we found that Quake 4 uses a texture for spotlight angular falloff, as shown to the right. If we perform a texture lookup on the green region which is clearly outside the falloff, mipmapping and clamping may cause the result to be the interval of the texture over the red region. Much better results are achieved if the culling is done directly on the texture coordinates, rather than using the more conservative texture lookup.

The texture lookup is one of our biggest sources of interval growth, and it is a clear candidate for improvements and future work. In fact, one of our main reasons for discarding affine arithmetic (apart from interval arithmetic being better suited for hardware) is that there is no obvious way to efficiently improve texture lookups with affine arithmetic, and that the texture lookups instantly break all linear dependencies. We did some preliminary tests using affine arithmetic for the arithmetic instructions, but most results were discouraging. The cull programs were either too simple with few dependencies, or contained texture lookups.

8 Conclusion

A long awaited feature in the GPU programming community is the ability to treat the depth buffer, color buffer, and possibly even stencil buffer contents as inputs to the fragment program. This would essentially remove the need for depth/stencil/alpha tests and blending, which are the last remaining parts of the fixed function pipeline.

According to Blythe [2006], the reasons why this functionality has yet to be implemented in hardware are, 1) pipeline hazards (a.k.a. read before write hazards), 2) multisampling issues, and 3) lack of culling due to the unpredictability of shader programs. Even though our PCU cannot help in the two first issues, it provides the functionality needed to solve the culling problem. For instance, it is not clear how to use hierarchical depth culling when an arbitrary depth test function is specified by the fragment program. With our PCU the culling comes automatically as long as the fragment program uses a `KIL` instruction for the depth test, and as long as the depth bounds of the triangle and depth buffer contents are given as inputs to the culling program.

The two remaining issues still are still open problems. Pipeline hazards has been, at least partially, solved by Donovan [2006] and Molnar [2006], who both introduce conflict detection units which ensure that no program threads are allowed to simultaneously access the same pixel elements. If such a conflict is detected, the conflicting threads are executed sequentially in the same order as they were issued, stalling the conflicting threads.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks to Jacob Munkberg, Petrik Clarberg, and Calle Lejdfors for inspiration and proofreading. All Quake 4 material courtesy of id Software.

References

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Transactions on Graphics, 22*, 3, 792–800.

AKENINE-MÖLLER, T., AND STRÖM, J. 2003. Graphics for the masses: A hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics, 22*, 3, 801–808.

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum, 23*, 3, 615–624.

BLYTHE, D. 2006. The direct3d 10 system. *ACM Transactions on Graphics, 25*, 3, 724–734.

COMBA, J. L. D., AND STOLFI, J. 1993. Affine arithmetic and its applications to computer graphics. In *SIBGRAPI 1993*, 9–18.

COOK, R. L. 1984. Shade trees. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 223–231.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Language Systems 13*, 4, 451–490.

DOGGETT, M., 2005. Overview of the xbox360 gpu. Keynote at *EUROGRAPHICS 2005*.

DONOVAN, W., 2006. Pixel load instruction for a programmable graphics processor. US Patent 7,091,979.

GREENE, N., AND KASS, M. 1994. Error-bounded antialiased rendering of complex environments. In *Proceedings of ACM SIGGRAPH 1994*, 59–66.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of ACM SIGGRAPH 1993*, 231–238.

HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Sampling procedural shaders using affine arithmetic. In *Proceedings of ACM SIGGRAPH 1998*, 158–176.

KEARFOTT, R. B. 1996. Interval computations: Introduction, uses, and resources. *Euromath Bulletin 2*, 1, 95–112.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, 149–158.

LOOP, C., AND BLINN, J. 2006. Real-time gpu rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics, 25*, 3, 664–670.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4, 43–55.

MCCOOL, M. D., WALES, C., AND MOULE, K. 2002. Incremental and hierarchical hilbert order edge equation polygon rasterization. In *Graphics Hardware*, 65–72.

MOLNAR, S., AND MONTRYM, J., 2006. Position conflict detection and avoidance in a programmable graphics processor using tile coverage data. US Patent 7,053,893.

MOORE, R. E. 1966. *Interval Analysis*. Prentice-Hall.

MOREIN, S. 2000. Ati radeon hyperz technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*, ACM Press.

MOULE, K., AND MCCOOL, M. D. 2002. Efficient bounded adaptive tesselation of displacement maps. In *Graphics Interface*, 171–180.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Graphics Hardware*, 41–50.

STAMMINGER, M., SLUSALLEK, P., AND SEIDEL, H.-P. 1997. Bounded radiosity — illumination on general surfaces and clusters. *Computer Graphics Forum 16*, 3, C309–C317.

TATARCHUK, N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (SI3D '06)*, 63–69.

URALSKY, Y. 2005. Efficient Soft-Edged Shadows Using Pixel Shader Branching. In *GPU Gems 2*. Addison-Wesley Professional, 269–282.

Instruction	Operation	Condition
`MOV` d, a	$d_i \leftarrow a_i$	
`MAD` d, a, b, c	$d_i \leftarrow [\min(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}) + \underline{c_i},$ $\max(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}) + \overline{c_i}]$	
`DP4` d, a, b	$d_x \leftarrow [\sum_i \min(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}),$ $\sum_i \max(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i})]$	
`RCP` d, a	$d_i \leftarrow [1/\overline{a_x}, 1/\underline{a_x}]$ $d \leftarrow [-\infty, \infty]$	$0 \notin a_x$ $0 \in a_x$
`RSQ` d, a	$d_i \leftarrow [1/\sqrt{\overline{a_x}}, 1/\sqrt{\underline{a_x}}]$ $d_i \leftarrow [\text{NaN}, \text{NaN}]$	$\underline{a_x} > 0$ $\underline{a_x} \leq 0$
`EX2` d, a	$d_i \leftarrow [2^{\underline{a_x}}, 2^{\overline{a_x}}]$	
`LG2` d, a	$d_i \leftarrow [\log_2 \underline{a_x}, \log_2 \overline{a_x}]$ $d_i \leftarrow [\text{NaN}, \text{NaN}]$	$\underline{a_x} > 0$ $\underline{a_x} \leq 0$
`MAX` d, a, b	$d_i \leftarrow [\max(\underline{a_i}, \underline{b_i}), \max(\overline{a_i}, \overline{b_i})]$	
`MIN` d, a, b	$d_i \leftarrow [\min(\underline{a_i}, \underline{b_i}), \min(\overline{a_i}, \overline{b_i})]$	
`SGE` d, a, b	$d_i \leftarrow 0$ $d_i \leftarrow 1$ $d_i \leftarrow [0, 1]$	$\overline{a_i} < \underline{b_i}$ $\underline{a_i} \geq \overline{b_i}$ otherwise
`SLT` d, a, b	$d_i \leftarrow 0$ $d_i \leftarrow 1$ $d_i \leftarrow [0, 1]$	$\underline{a_i} \geq \overline{b_i}$ $\overline{a_i} < \underline{b_i}$ otherwise
`FLR` d, a	$d_i \leftarrow [\lfloor \underline{a_i} \rfloor, \lfloor \overline{a_i} \rfloor]$	
`NEG` d, a `SAT` d, a	$d_i \leftarrow -[\overline{a_i}, \underline{a_i}]$ $d_i \leftarrow [\max(0, \min(1, \underline{a_i})), \max(0, \min(1, \overline{a_i}))]$	

Table 2: *The arithmetic and conditional expressions of our minimal instruction set, named in accordance to the ARB_fragment_program extension. We use d for the destination register, and a, b & c for source registers. Note that `NEG` and `SAT` are not actual instructions. They refer to the optional negation of source registers, and the optional saturation of the result. Also, the `DP3/DPH` are left out since they are simple modifications of the `DP4` instruction. Note that whenever i is used above, it implies componentwise computations, i.e., $i \in \{x, y, z, w\}$, and also we have omitted the hats (ˆ) to avoid cluttering.*

A Interval Arithmetic Instruction Set

In this section we introduce an instruction set operating on intervals. This includes arithmetic operations, conditionals, and texture lookups. We base the native instruction set of our PCU on a subset of the fragment program instructions defined by the OpenGL ARB.

A.1 Arithmetic and Conditional Instructions

Arithmetic and conditional operations have been thoroughly studied in interval arithmetic [Moore 1966; Kearfott 1996], and we base our instruction set on those findings. The instructions and their corresponding operational behavior are summarized in Table 2.

A.2 N-Dimensional Texture Lookups

The interval instructions for performing N-dimensional texture lookups are inspired by Moule and McCool's [2002] approach, originally used for displacement map subdivision. The general idea is to provide an efficient way of computing the bounding interval of the texture data over a given area[2]. The remainder of this section

[2]Interval-based texture example: assume we render a fence as a quad, using alpha testing and a texture to represent the geometry. It would then make sense to cull tiles where $texture_\alpha = 0$ over the whole tile. This can be done using the interval-based texture lookup.

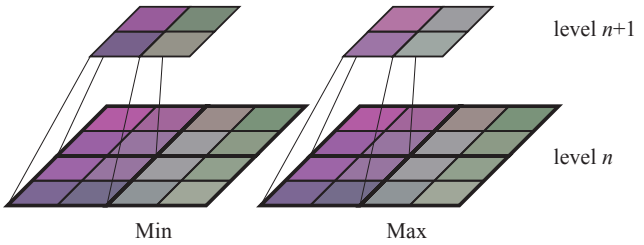


Figure 5: *The mipmap pyramids used for conservative texture lookups. We compute two pyramids, where each texel contains the minimum and maximum values respectively, of the corresponding four texels in the mipmap level directly below.*

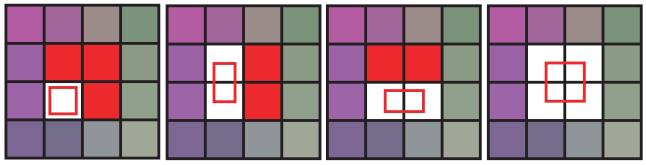


Figure 6: *The four different lookup patterns of conservative texture lookups. Mipmap selection makes sure that the texture coordinate interval (red rectangle) is never more than one texel wide. Therefore, only the four cases shown in the image are possible. Since we always read a block of 2×2 texels (in the spirit of linear filtering) we exclude the texels shaded in red from the final result.*

will only consider two-dimensional textures, but generalization to a higher dimension is straightforward.

We initially compute two mipmap pyramids for each texture that is subject to interval-based texture lookup. As shown in Figure 5, each element in a mipmap is computed as the component-wise minimum or maximum value of the four corresponding texels immediately below it in the pyramid. The final result can be seen as a mipmap pyramid of bounding intervals. This type of pre-computation can easily be handled by the driver, similar to how standard mipmaps are auto-generated.

When performing a texture lookup, we wish to compute the bounding interval of the texture data over an axis-aligned bounding box, which is the texture coordinate interval. First, we compute an appropriate mipmap level as:

$$\lambda = \left\lceil \log_2 \left(\max(\overline{t_x} - \underline{t_x}, \overline{t_y} - \underline{t_y}) \right) \right\rceil \tag{5}$$

where $\hat{\mathbf{t}} = (\hat{t}_x, \hat{t}_y)$ is a two-dimensional interval of the unnormalized integer texture coordinates (i.e., they include the dimensions of the texture). These are conservatively rounded so that $\underline{t_i}$ is floored and $\overline{t_i}$ is ceiled for $i \in \{x, y\}$.

When transformed to this mipmap level, $\hat{\mathbf{t}}$ will never be more than one texel wide in any dimension, and always least $1/2$ texels wide in the widest dimension. Thus, we get four possible cases of texture coordinate intervals as illustrated in Figure 6. Here, we deviate slightly from Moule and McCool [2002] who used these cases to determine how many texels they need to sample. Instead, we always sample a square of 2×2 texels with the lower left corner at the texel of $(\underline{t_x}, \underline{t_y})$, in the access scheme used for normal linear interpolation. The result of the texture lookup is then computed as the bounds of the colors of the texels that actually overlap the texture coordinate interval. That is, we discard the texels shaded in red in Figure 6. Since the mipmap transformed $\hat{\mathbf{t}}$ will be rounded to the nearest integer coordinates, this overlap test can be implemented very efficiently by comparing just the final bit.

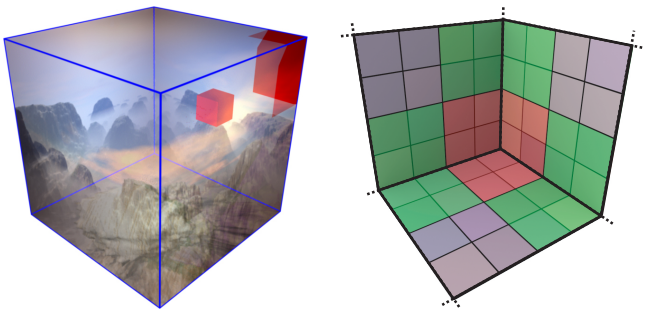


Figure 7: *Left: The cube mapping problem. A texture coordinate interval (the red box) may map to several sides on the cube (the projection of the red box), but current hardware only allows one side at a time to be accessed. Right: Min-Max mipmap generation for interval-based cube mapping. Only a small part of the cubemap is shown here. We use texels from both sides of an edge (the green regions), and texels from three sides in the corners (the red region).*

It can be shown that this texture lookup process is conservative with respect to filtered texture lookups, such as bi-linear, tri-linear and anisotropic filtering, as long as the filtered texture lookups compute derivatives using finite differences, and as long as the texture filter does not extend outside the area spanned by the derivatives. The texture lookup also natively supports all different kinds of wrapping modes, such as clamping and repeating. The appropriate wrapping mode can simply be applied to the interval coordinates, after mipmap level computation, to get the expected result. Our texture lookup process is essentially as costly as a normal tri-linearly filtered texture lookup. The biggest differences are that we need to be able to sample from the same level in two different mipmap pyramids rather than two adjacent levels, and that we compute the final result as a bounds rather than using linear interpolation.

Discussion If more texture units are available, it is possible to improve the bounds of the interval texture lookup. The normal texture lookup assumes that we can read a block of 2×2 texels at a time. If we have enough hardware resources to read a block of 4×4 texels instead, then we can move one level down in the mipmap hierarchy and get a more accurate result.

Another important observation is that we only need to create the mipmap levels that are actually used in the cull program. This optimization is particularly important for algorithms taking place in screen space such as order independent transparency (see Section 6). In this case, we know beforehand that we only need the texture at its base level and the mipmap level that corresponds to a tile on the screen. Note that such tile information is already available in modern hardware and can be read "for free". The minimum and maximum depth values can, for instance, be found in the hierarchical depth culling unit. It is also possible (but less likely) that the min and max colors are already computed for compression purposes, otherwise we need to compute them. Extensions for rendering to the base and tile mipmap level of a texture would greatly accelerate screen space algorithms.

A.3 Cube Map Lookups

Cube maps present more of a challenge than normal texture lookups. Normally, a cube map is accessed through a vector coordinate (x, y, z), and thus we get a three-dimensional coordinate interval. As can be seen in the left part of Figure 7, we want to

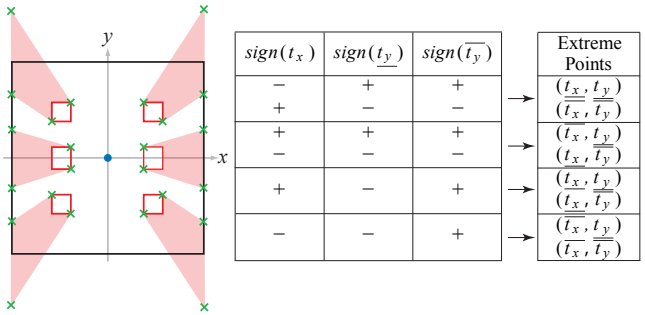


Figure 8: *Our method for projecting the bounds of a texture coordinate interval on the cube map. Here x is the major axis, and we want to find the texture coordinate interval when projected on any of the x-sides of the cube map. The figure illustrates the six possible cases of texture coordinate intervals, and the projected extreme points. The table summarizes the different cases of extreme points and show how they can be determined using signs.*

project that interval onto the cube map and compute the interval of the texture data under the projected area.

There are two problems with this approach. First, it is expensive to project a cube onto the shape of another cube. Second, hardware is typically designed with the restriction that each texture lookup may only access a single side of the cube. Since we strive for as modest hardware modifications as possible, we propose an alternate method which trades accuracy[3] for a simpler algorithm.

We compute the min/max mipmap pyramid for the cube map, using the same approach as for two-dimensional textures. However, near edges and corners of the cube, special treatment is necessary as shown to the right in Figure 7. For the edges, we compute the mipmap color as the min or max of four texels on both sides of the edge, and for the corners we compute the mipmap color as the min or max of four texels on all three sides emanating from that corner. Texels on opposite sides of edges will therefore share the same colors in higher mipmap levels. Similarly, the three texels in a corner will also share a common color. It should be noted that the highest level mipmap will contain the min and max value over the full cube, as expected.

We can now use this mipmap pyramid to do conservative cube map lookups with accesses to only one side of the cube. First, we compute the interval-based equivalent of the major axis. Given a texture coordinate interval $\hat{\mathbf{t}} = (\hat{t_x}, \hat{t_y}, \hat{t_z})$, we define the major axis, i, as the axis where $\underline{t_i}$ and $\overline{t_i}$ have the same sign, and where $\min(|\underline{t_i}|, |\overline{t_i}|)$ is maximized. This is essentially the axis where the texture coordinate interval does not contain the origin, and lies nearest to a cube map face. If $\underline{t_i}$ and $\overline{t_i}$ have different signs over all axes, then we cannot find a major axis. However, this can only happen if the origin lies within the texture coordinate interval. In this case, the texture coordinate interval will project onto the entire cube map. We can easily handle this by choosing the highest mipmap level, and sample an arbitrary cube map face.

Once we have found a major axis, we conservatively project the texture coordinate interval on the corresponding side of the cube map. The projection is done by projecting the bounds of each of the two remaining axes separately. Figure 8 shows such a projection, where x is the major axis, and y is the axis for which we want to project the bounds. The figure shows the six possible cases of texture coordinate intervals (note that no interval may cross the y-axis since the x-axis would not be a major axis in that case), and the extreme points we have to project to compute the bounds. Fortunately, it is very easy to determine which these extreme points are. It is sufficient to look at the signs of the texture coordinate interval, as shown in the table in Figure 8.

We project the extreme points for the remaining two axes to form a two-dimensional projected coordinate interval. This interval is used to compute a mipmap level and perform a two-dimensional texture lookup, identically to the method described in Section A.2.

It is possible to show that this algorithm is conservative because of the information-bleeding during mipmap generation. Furthermore, it is computationally inexpensive. Finding the major axis, and projection can be expected to be twice as costly as a normal cube mapping implementation, which is reasonable considering we use intervals. In addition we need the tabulated function from Figure 8, which is already very inexpensive.

[3] In the sense of interval width. The algorithm is still fully conservative.