Real-Time Shadows

Broderick Arneson

Unversity of Alberta, Edmonton, AB, Canada
broderic@cs.ualberta.ca

Abstract. We investigate two different techniques for the rendering of real-time shadows in hardware: shadow mapping and stenciled shadow volumes. A renderer capable of viewing Quake3 map/model files is used as a base from which both shadow algorithms are added. Shadow mapping performs well, but visual artifacts are difficult to completely eliminate. The shadows can also be blocky, even if the texture-size is quite large. Shadow volumes give nice shadows, but the overhead of the stencil write passes can severely affect the framerate. If optimized, the shadow volume technique is probably more suitable for this environment.

1 Introduction and Motivation

Shadows are an important part of real-time graphics since they add depth and realism to a scene. Id Software recently released Doom3, the first real-time 3D game to use fully dynamic lighting instead of precomputed lightmaps. These are done with the shadow volume method. While Doom3 requires a fairly powerful machine to run at respectable frame-rates, it shows that real-time shadows are possible on todays hardware.

The purpose of this paper is to investigate two different methods of generating real-time shadows and to compare their speeds and visual quality. The first method is an image based algorithm known as shadow mapping. Being image based, it has the advantage of not requiring any knowledge of the world geometry, but it suffers from aliasing effects. The shadow volume method attempts to avoid these aliasing errors by generating geometric shadow volumes; but this of course results in more overhead and necessitates dealing with the world's geometry.

In section 2 we discuss the base renderer, which we used as a starting point before adding both shadow algorithms, and our implementations in sections 3 and 4. Finally, we compare the two methods in section 5.

2 Quake3 Renderer

We chose to use Quake3 as a test-bed for several reasons. Firstly, the abundance of maps and models available for use during development and testing eliminates the need for us to tediously construct our own environments. Secondly, it is a relevant real-world application — Quake3 was extremely popular in its time and the game engine itself is the base for many popular games recently released (id software licenses its engines to other game companies). Hence, while the environment is not as complex as some newer games, it still offers enough complexity to make real-time shadows a challenging problem. Thirdly, the specifications of the map and model files are easily available online, so constructing a simple viewer was quite easy. The following are some properties of our renderer and of Quake3 maps/models in general.

2.1 Models

Each map is composed of one or models. The first model is always the static world, subsequent models (if any) represent movable parts of the world (eg: doors that open and close, lifts that rise, etc). We render only the static world for simplicity.

2.2 BSP-trees

The world is subdivided using a BSP-tree. Each node and leaf has a bounding box which is used to cull nodes and leaves that are not in the view frustum. Adjacent leaves can be grouped into *clusters* and for each cluster in the world, information on which other clusters are visible is stored in a binary look-up table. This precomputed visibility information, along with the frustum tests, are the only methods of surface culling employed in our renderer.

Not every face in the world is used when creating the BSP-tree; some faces can be marked as "decorative" and these faces are not considered during the construction of the tree. This greatly simplifies the

BSP-tree, at the cost of allowing faces to be in several leaves at once as well as allowing non-convex leaves. Non-convex leaves are not a problem for us however, since we are not relying on the BSP-tree to order surfaces for rendering.

The world is not guaranteed to be a 2-manifold, that is, each edge in the world can be shared by more than two faces. In all cases however, the extra faces are "inside" the geometry and hence are not visible. It is also possible for some faces to overlap each other, although this usually occurs in areas inaccessible to the player.

2.3 Faces

There are four types of faces in Quake3 worlds: polygons, meshes, patches, and billboards. Polygons are convex and planar, and are rendered using a triangulation of the face given in the map file.

Meshes are simple closed triangle models (these **are** 2-manifolds). These are used to represent objects like teleporter stations and torch holders that require fine detail, but are still part of the static world.

Patches refer to quadratic Bezier patches. These could be tessellated on-the-fly with varying level of detail, but we tessellate them just once at startup for simplicity. After tessellation they are internally represented as meshes and so the code to render them is the same.

Billboards are quads that always face the viewer — these are ignored.

2.4 Shaders and Lightmaps

Each face in the world has a shader (not to be confused with hardware vertex and fragment shaders) as well as a precomputed lightmap. Shaders are used to define the surface texture (or textures: several can be combined with various blending methods). For simplicity, we use only the first texture specified in the shader as the face's texture. The face's lightmap is used to modulate the face's texture; and so is our source of ambient lighting. We render both in one pass using the `ARB_multitexture` OpenGL extension.

2.5 Triangle Models

Player models and other dynamic objects are loaded from md3 files (this is a simple format the game designers created specially for this purpose). Each model is composed of several closed triangle meshes, and each mesh is one frame of animation. Typical models are composed of a few hundred triangles and can have up to several hundred frames of animation. Each model can be "skinned" using different textures, but again, we use only the first skin specified for simplicity.

Specifying a lightmap for these models would not make sense since they are not constrained to any one position; thus specifying the ambient light level for each model poses a problem. For simplicity we render each model at a constant, moderate, light level; and for most situations this looks fine. After all, we are most interested in the dynamic lighting of world.

2.6 Statistics and Speed

Most Quake3 maps have from 5000 to 15000 faces, and at any given time there are usually around 10k to 25k triangles visible to the viewer (that is, not culled using the precomputed visibility tests or frustum culling). The world is rendered in a single pass using two texture units, as mentioned above.

Our test machine is an AMD64 3200+ with 1G of RAM and an ATI Radean 9800 Pro (256-bit) graphics card with 128MB of texture memory. At 640x480, typical frame-rates range from 120fps to 250fps on this machine. There are usually around one hundred separate textures in use in a level at once, with the largest texture size being around 256×256 (most texture are much are smaller). This means with 32-bit textures and mipmapping enabled, at most 50MB of texture memory is used, and typically only around 30MB or in practice.

This gives us plenty of room to add real-time shadows!

3 Shadow Mapping

Shadow mapping is a fairly straightforward idea: we draw the scene from the light's point of view, only caring about the depth values at each pixel. We call this created image the *shadow map*, and will use it on a later pass to determine which pixels are in shadow. This check is done by projecting the point in question from the camera's view-space into the the light's clip-space, giving us a specific pixel in the shadow map. We compare the distance between this point and the light and the value in the shadow map, if the value is greater the point must be in shadow. That is really all there is to it. For more details on shadow mapping refer to [1] [2] [4].

The advantages of shadow mapping are that we do not need to know anything about the geometry in order to get shadows; this is important because for really complicated objects, or really complicated scenes, the geometry may be very difficult or expensive to work with. At the same time, shadow mapping suffers from precision errors and aliasing artifacts.

3.1 Implementation

We store our shadow buffers as 24-bit depth textures using the `ARB_depth_texture` extension. OpenGL's built-in texture coordinate generation is used to project the camera's eye-space coordinates into a light's clip-space coordinates. We use the `ARB_shadow` extension to compare the generated r coordinate to the shadow buffer value at s and t. Values that pass this test create an intensity result with an alpha value of 1.0; we use an alpha test to determine whether the pixel is in shadow or not. This approach is similar to the one outlined in [5]. What follows is a discussion of the problems encountered during implementation.

P-buffers We would like to render the shadow map into an offscreen depth texture, however, the ATI card I wrote this on (a Radeon 9800) does not support the `ARB_pbuffer` extension; this extension would allow hardware accelerated rendering to a texture in a straightforward manner. So, another choice is to create an offscreen context in X (or in Windows) and render to this, then copy the depth values from this context's depth buffer into the shadow map texture. An easier solution, however, is to simply render to the current depth buffer and copy the depth values to the depth texture without creating any offscreen contexts. The `ARB_depth_texture` extension does this copy completely on the card, and so the copy operation is reasonably fast.

This approach works well when there is only one light, but when several lights are visible we must copy the results for each light into separate depth textures before we can render the unlit world. That is, instead of the algorithm being:

```
- Render unlit world
  For each light that affects the scene:
   - Render light's view into pbuffer
   - Copy pbuffer into shadow map
   - Draw lit world with additive blending
```

(if we can render directly into a texture then the render and copy steps would be combined into one), we actually do the following:

```
  For each light that affects the scene:
   - Draw light's view into screen
   - Copy depth values to shadow map
- Draw unlit world.
  For each light that affects the scene:
   - Draw lit world with additive blending
```

This of course takes more texture memory than a single pbuffer. I chose this method to avoid using a platform-specific technique to obtain an offscreen context. Also, as mentioned in section (BLAH), only around 30MB of texture memory is being used to store lightmaps and textures, hence sparing a few MBs for shadow maps seems a justifiable cost since modern graphics cards typically have upwards of 128MB.

Visual Artifacts One problem with shadow mapping with 2D shadow maps is that lights are by default spotlights; what happens if a point is not in the light's FOV? The `ARB_shadow` extension culls points with negative r coordinate: this means points behind the light are always in shadow. However, since the s and t coordinates are clamped to within $[0, 1]$ (if we do not clamp then they wrap around, which is worse), points in front of the light, but outside its FOV, get shadow values at the corresponding edges of the shadow map.

There are several ways to get around this. One way is to create two other texture coordinate generating operations, one for each of s and t, that only pass if the values are between $[0, 1]$. This is similar to a technique used in projective texture mapping to cull back-projections [3].

Another, simpler approach, is to specify the light's view frustum planes as extra OpenGL clipping planes during the lighting pass. This has the advantage of reducing the amount of geometry that is rasterized for each light, but also introduces visual anomalies like in Figure 1. Note these are not errors introduced from the shadow map lookup: the depth values of some fragments are actually different from the previous unlit pass. This is because triangles that touch the light's clipping plane are clipped, and consequently result in different view-space polygons being projected and rasterized. We have solved one problem only to introduce another!

Actually, we have not really introduced another problem: as alluded to above, it is possible for small floating point errors to result in some points being in

shadow when they should not be, and so this problem was there all along. Again, see Figure 1. What we really need to do is during the lit pass, offset the polygons such that their depth values are slightly less than what are in the depth buffer.

The most obvious way to do this is to simply move the viewer a little closer to the scene during the lit pass. This works some of the time, but anomalies still occur along polygons sloping away from the viewer. That is, the slope of a polygon relative to the viewer determines how much of an offset we need to use.

Fortunately, OpenGL has a built-in feature to deal with just this problem. We use `glPolygonOffset()` to offset the depth values — this takes into account the polygon's slope — before comparing them with the current depth buffer. This eliminates most visual anomalies. The amount we need to offset varies, and depends on the light's angle and field of vision. Also, if more than one light affects a fragment, it is necessary to use larger and larger offsets for each subsequent light to avoid anomalies.

Omni-directional Lights There are several ways of handling omni-directional light sources, but we deal with them in the simplest way possible: which is not at all! Instead of one light with a 360-degree FOV, we use six lights with 90 degree FOVs. This works reasonably well but is a little inefficient. Note that using fewer lights with larger FOVs results in inaccurate shadows.

Another way of handling omni-directional lights is to generate 3D textures instead of several 2D textures — we decided to go with the simplest approach since this technique is quite different from our initial implementation.

Aliasing Artifacts Shadow map resolution affects the quality of the shadows produced. Figure 2 shows the same scene using several different shadow map resolutions. We have found that a shadow map resolution of 512x512 gives reasonably sharp shadows while still offering acceptable performance.

A technique known as Percentage Closer Filtering (PCF) can be used to smooth out shadow blockiness — note that normal linear interpolation is *not* a valid solution since it is not meaningful to interpolate between depth values in the shadow map, as discussed in [4]. PCF instead takes several samples and averages the *result*. That is, instead of using only one comparison to determine whether a pixel is in shadow, we take nine (or any other number) samples from different parts of the pixel and use the percentage of the number of samples in light to modulate the light intensity at that pixel. For example, if five of the nine samples are in shadow, then this pixel would receive only 4/9th's of the light.

OpenGL does not offer a way to do this automatically, and for simplicity (and because of time constraints!) we have chosen not to implement it. One way to do it is to have a fragment shader program perform PCF during the lighting pass.

3.2 Results

Using shadow mapping without PCF is somewhat unsatisfactory. From a distance the shadows may appear fine, but up close the blockiness is apparent. In a game type situation, where image quality is important, PCF would definitely be a requirement. However, as pointed out in [**?**], even with PCF and very high resolution shadow maps will always suffer when the camera faces the light (the "duelling frustums effect"). Another downside is the biasing values need for `glPolygonOffset` vary from light to light — this can make the design of game levels tedious and frustrating.

Performance is decent on my hardware. With 2 to 10 lights in a scene, framerates of around 30fps are very possible, and can sometimes still exceed 100fps. The size of the shadow map texture affects performance, but the area of the screen visible to the light seems to be the the dominant factor.

In all, if the hardware can handle large texture sizes while doing PCF, the image quality should be good; if the artifacts can be tamed.

4 Shadow Volumes

Shadow volumes are another way to generate real-time shadows. For each light, and each silhouette edge with respect to the light (defined as an edge that borders a triangle that faces the light and a triangle that faces away from the light), we extrude the edges to infinity, creating a four-sided face for each edge. These faces define the shadow volumes. We now draw these faces into the stencil buffer from the camera's point of view, adding and subtracting stencil values as appropriate. If we are using the Depth-Pass technique, we increment the stencil buffer for front-faces that pass the depth-test, and subtract for back-faces that pass the depth-test. If we are using Depth-Fail (also

known as Carmack's Reverse), we increment on back-facing faces that fail and decrement on front-facing faces that fail. Any pixel whose stencil value is 0 is then lit by the light, otherwise it is in shadow.

Depth-Pass works as long as the camera does not enter the shadow volume — otherwise the shadows may not be computed correctly. Depth-Fail *does* work if we enter the shadow volume, but it requires the volumes to be closed, a condition not imposed by Depth-Pass. To close a shadow volume, we include the front-facing geometry as a front cap and extrude them to infinity, while flipping their direction, to obtain a back cap. We use the Depth-Fail technique since not being able to enter a shadow volume is a serious limitation.

The main challenge in getting shadow volumes to work is ensuring that any closed volumes are air-tight, and at the same time, rendering as few faces as possible for speed. The advantages over shadow maps are crisp, sharp shadows with no aliasing or other artifacts. The disadvantages are the extra processing time needed to compute and render the volumes.

4.1 Implementation

There were several obstacles that needed to be overcome in order to get shadow volumes working correctly in the Quake3 environment. We discuss these in this section.

Infinite Volumes We must be able to extrude the shadow volume quads to infinity: it is not enough to extrude them by a large finite distance since it would then be possible for the shadow to terminate prematurely (this can happen if a lightsource is close to an occluder). Given a light at position $L = (x_L, y_L, z_L, w_L)$ and a point $P = (x_P, y_P, z_P, w_P)$, we extrude P to P' with the equation $P' = (x_P - x_L, y_P - y_L, z_P - z_L, 0)$.

We now need to ensure that the camera's far-plane does not clip the points at infinity (this is especially important when using Depth-Fail). This is accomplished by moving the camera's far-plane itself to infinity. Hence our new projection matrix becomes

$$P_{new} = \lim_{f \to \infty} P,$$

where f is the distance to the far-plane and P is the standard OpenGL projection matrix (which takes f as an argument). [8] explains this in detail. It is easy to verify that points at infinity are not clipped by P_{new}.

We are now set to render the shadow volumes, all we need is to do is compute them first.

Volumes and Meshes Computing the shadow volume quads for meshes is easy (by mesh we mean the static meshes of the map, any tessellated Bezier patches, and md3 models). We find all the unique edges for each mesh at startup, storing which triangles border each edge. We also compute and store the normals for each triangle.

To compute the shadow volumes for a mesh given a light L, we first need to determine the silhouette edges. The simplest way to do this, and the way we do it, is to first compute which triangles face L. After we have done this, we run through the edges and mark as a silhouette edge any edge which borders exactly one front-facing triangle.

We then extrude all the silhouette edges and add the front-facing triangles and their flipped extrusions as the front and back caps.

Volumes and the World There would be no problems if the world was a 2-manifold. However, it is not, and this fact caused me many hours of debugging fun trying to figure out why nothing was working.

Using Depth-Fail and extruding every edge eliminates any errors and fixes the problem; this is terrible solution however, since most extruded edges are redundant. To improve on this, while preprocessing the edges we keep track of which faces are on each edge as with meshes (but now there could be be many more than two). We skip the extrusion of an edge if it is shared by exactly two front-facing polygons, otherwise we force we extrude. For edges shared by more than two faces, we play it safe and extrude all of them (some of which are redundant). Luckily, these cases are rare. This whole approach is obvious in retrospect, but took awhile to get right.

Rendering Shadow Volumes We render the shadow volumes for each light in two passes. We draw only back-facing polygons in the first pass (incrementing on depth fails as per Depth-Fail) and front-facing polygons in the second pass (decrementing on depth fails). These passes render into the 8-bit stencil buffer only. It is possible to do both in one pass using the `GL_EXT_stencil_two_side` and `GL_EXT_stencil_wrap extensions`, but my card does not support these.

4.2 Results

The visual quality of the shadows produced using shadow volumes is quite good. They were more work to implement, however, since messing around with the world geometry can be a tricky business.

Framerates depend highly on the number of shadow quads rendered. With two or three lights in a scene, it is possible to have several thousand sides and several thousand caps per light, so the two stencil passes per light quickly dominate. If lights are kept small (so that they can be culled from the scene) then performance should be acceptable. On my machine, with around 3-4 lights visible at a time, framerates of around 30-40fps are standard. If more lights are potentially visible, this can drop to around 20fps or so.

One problem is that the Quake3 world was not designed with the requirements of stenciled shadow volumes in mind, since the extra edges we need to extrude for correct shadows only hurt performance.

Another problem is that we don't switch between Depth-Fail and Depth-Pass like we should. It is simple to check if we are potentially inside a shadow volume, and therefore if we need to use Depth-Fail and add the caps. If we performed this check on a per leaf basis, it should be possible to use Depth-Pass on leaves where it is safe to do so; and thus we could reduce the number of shadow quads needed. The world would also need to be a 2-manifold for Depth-Pass to work correctly on it, however.

In any case, there is definitely still room for improvement.

5 Conclusion

I am partial to using shadow volumes for this environment. The visual artifacts obtained in shadow mapping are controllable, but can be painful to deal with—and to look at. Shadow volumes just create much higher quality shadows, period. In addition, it seems possible to speed up the shadow volume method as it currently stands, but I see no real way to increase the performance for shadow mapping. For these reasons, the shadow volume method seems better suited for real-time shadows in an environment like that of Quake3.

References

1. E. Haines and T. Möller, "Real-Time Shadows", *Games Developers Conference*, March 2001.

2. A. Woo, P. Poulin, and A. Fournier, "A survey of Shadow Algorithms", *IEEE Computer Graphics and Applications*, vol. 10, no. 6, pp. 13-32, November 1990.

3. C. Everitt, "Projective Texture Mapping". White paper attainable at: http://developer.nvidia.com/object/Projective_Texture_Mapping.html

4. C. Everitt, A. Rege, C. Cebenoyan, "Hardware Shadow Mapping", http://developer.nvidia.com/object/hwshadowmap_paper.html.

5. "Shadow Mapping Tutorial", http://www.paulsprojects.net/tutorials/smt/smt.html.

6. "Carmack on Shadow Volumes", http://developer.nvidia.com/attach/6832.

7. "Practical and Robust Shadow Volumes", http://developer.nvidia.com/object/robust_shadow_volumes.html.

8. E. Lengyel, "The Mechanics of Robust Stencil Shadows", http://www.gamasutra.com/features/20021011/lengyel_01.html

9. "Q3A Shader Manual", http://www.heppler.com/shader/.

10. n "Description of MD3 Format", http://www.icculus.org/homepages/phaethon/q3a/formats/md3format.

11. "Unofficial Quake 3 Map Specs", http://graphics.stanford.edu/ kekoa/q3/.

12. "Rendering Quake 3 Maps", http://graphics.cs.brown.edu/games/quake/quake3.html.

13. "Cake: a Quake 3 Map Viewer", http://www.calodox.scene.org:8080/morbac/cake/.

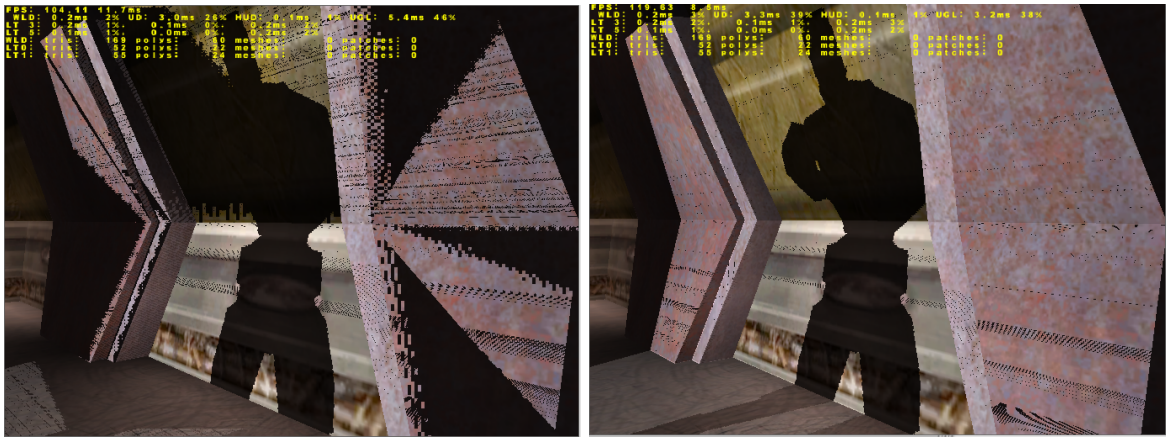

Fig. 1. Anomalies from clipping planes (left); without glPolygonOffset() (right)

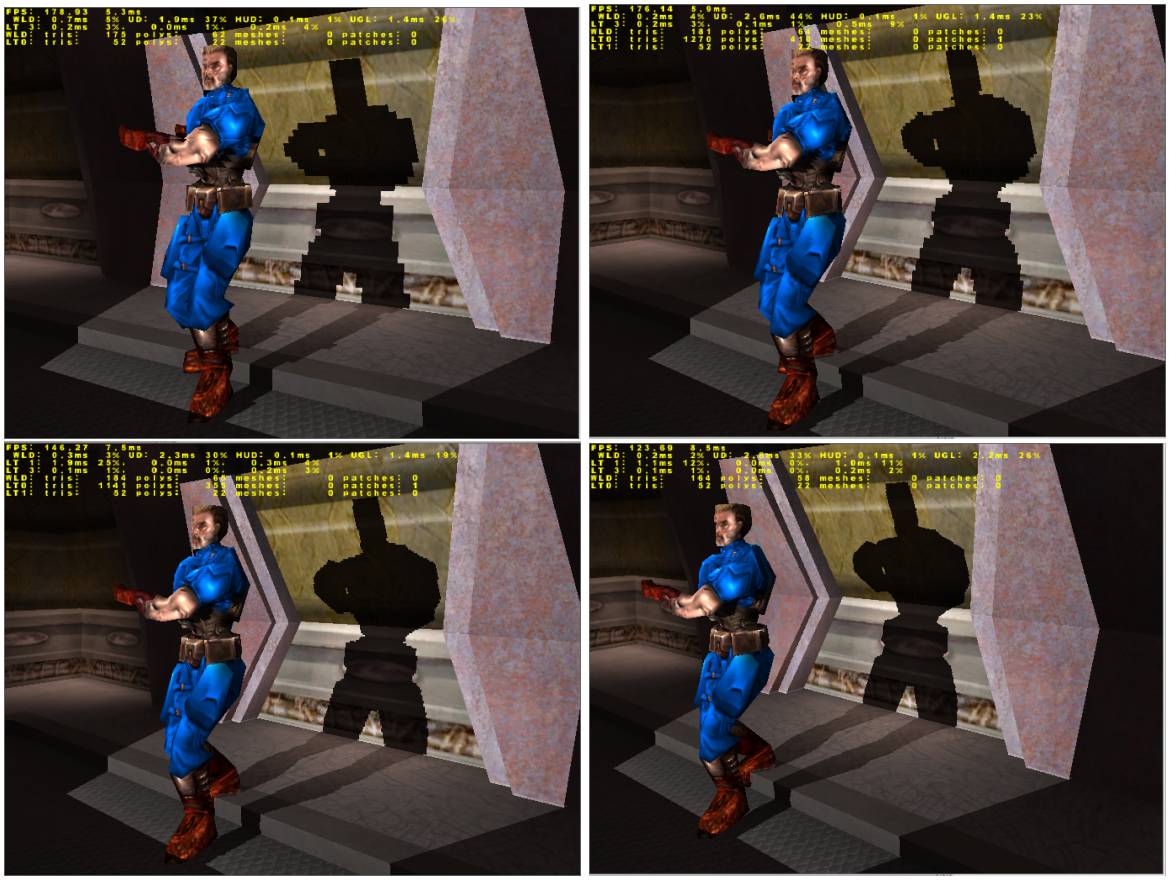

Fig. 2. Different shadow map resolutions: 64x64, 128x128, 256x256, 512x512.