CSM Scrolling:
An acceleration technique for the rendering of cascaded shadow maps

Abstract

This talk will explain how a bitmap-scrolling technique can be combined with a shadow map caching scheme to significantly increase the performance of real-time cascaded shadow mapping in games. The two systems integrate well into the standard model of cascaded shadow mapping, and take advantage of temporal coherence to preserve much of the rendered shadow map information across frames. The technique is well-suited to current games consoles, and will ship in “Overstrike”, the forthcoming title by Insomniac.

1. Introduction

We build on the standard technique of Cascaded Shadow Maps (CSM), described in [Engel 06]. A linear depth buffer for shadow maps will be assumed throughout, with near and far planes corresponding to depths of 0.0 and 1.0 respectively. We will use the term ‘sub-frustum’ to describe the subsection of the camera’s frustum of view over which a given CSM map will be used.

In a typical game scenario, we can make the observation that the contents of the CSM buffers exhibit a great deal of coherence across frames. Usually, the direction of sunlight will be constant (or can be treated as such, most of the time) and many scene objects remain stationary, the only entities in motion being the camera and a typically small subset of the scene geometry.

If the camera is static, scene elements such as the ground and buildings will generally remain unchanged in the shadow maps and only relatively smaller objects move and need to be redrawn into the maps. This suggests that the CSM process can benefit from having the static portions cached across frames, so that only small portions need drawing from scratch.

If, on the other hand only the camera is in motion, since the shadow maps are constrained to follow sections of the camera’s view frustum, the entire map contents will be changing from frame to frame. However, the changes occur in a predictable way which can be exploited via a scrolling technique to preserve much of the map contents, again reducing the amount of geometry in need of redrawing.

In general, both the camera and other scene elements will be in motion, and a combination of the caching and scrolling techniques can be used to accelerate the generation of the maps each frame.

2. Shadow map caching

CSM scrolling builds on a more general technique for caching shadow maps, described here. Consider a real-time lighting system in which a shadow map is generated for each light. In preparing a light's map for a given frame, typically a scene query will be performed to determine which scene objects are 'visible' to the light and so must be drawn into the map. This query will usually test scene objects for overlap with the light's region of influence, and perhaps for occlusion from the light's point of view. Objects passing the test are submitted for rendering into the map. We can also flag each scene object as either active or static, where to be regarded as static the object must fulfill certain criteria, such as remaining stationary for a prescribed interval of time – 10 seconds, for example. (A many-frame time interval is appropriate here, long enough to ensure that the change of state between active and static is comparatively rare.)

Sometimes, the scene query will return the same list of objects as it did on the previous frame. If all of these objects are flagged as static, and if furthermore the shadow-casting light hasn't moved, then clearly the shadow map will be identical to the one from the previous frame. If we cached the map in memory on the previous frame, we can simply reuse it as-is on the current frame.

Often, though, something in the map will change – say a character is walking in the path of the light. It may still be that a substantial portion of the rest of the map will remain the same. This latter portion will include at least all of those objects flagged as static, so it will still be beneficial to cache a version of the map containing just the static objects, and generate a finished shadow map from it by first copying it into a working memory area and then rendering the active objects onto it. If the list of active objects is small, this may save a significant amount of rendering work, at the cost of the persistent memory set aside for the cached map. We lower this cost somewhat by utilizing a 16-bit depth format for maps.

A pseudocode summary of the shadow map caching technique follows, and figure 1 summarizes the process outlined in the pseudocode.

Note that the occlusion tests for determining the list S should only make use static scene objects in the role of occluder. We want to ensure that a member of S, once occluded, remains occluded, and active occluders are clearly unsuitable for this.

```
// to generate working shadow map W, having query volume V,
// while maintaining cached shadow map C

invalidate C
S={}

for each frame
{
  if light moved
    invalidate C

  for each scene object
    flag as static or active

  let S'=S

  query scene to find:
  S = {static objects intersecting V and not occluded}
  A = {active objects intersecting V and not occluded}

  if C is valid
  {
    if any member of S' is no longer static or has been deleted
      invalidate C
  }

  if C is valid
  {
    // reuse cached map
    let ΔS = S-S'
    render ΔS to C
  }
  else
  {
    // redraw cached map
    clear C
    render S to C
  }

  flag C as valid
  copy C to W
  render A to W
}
```

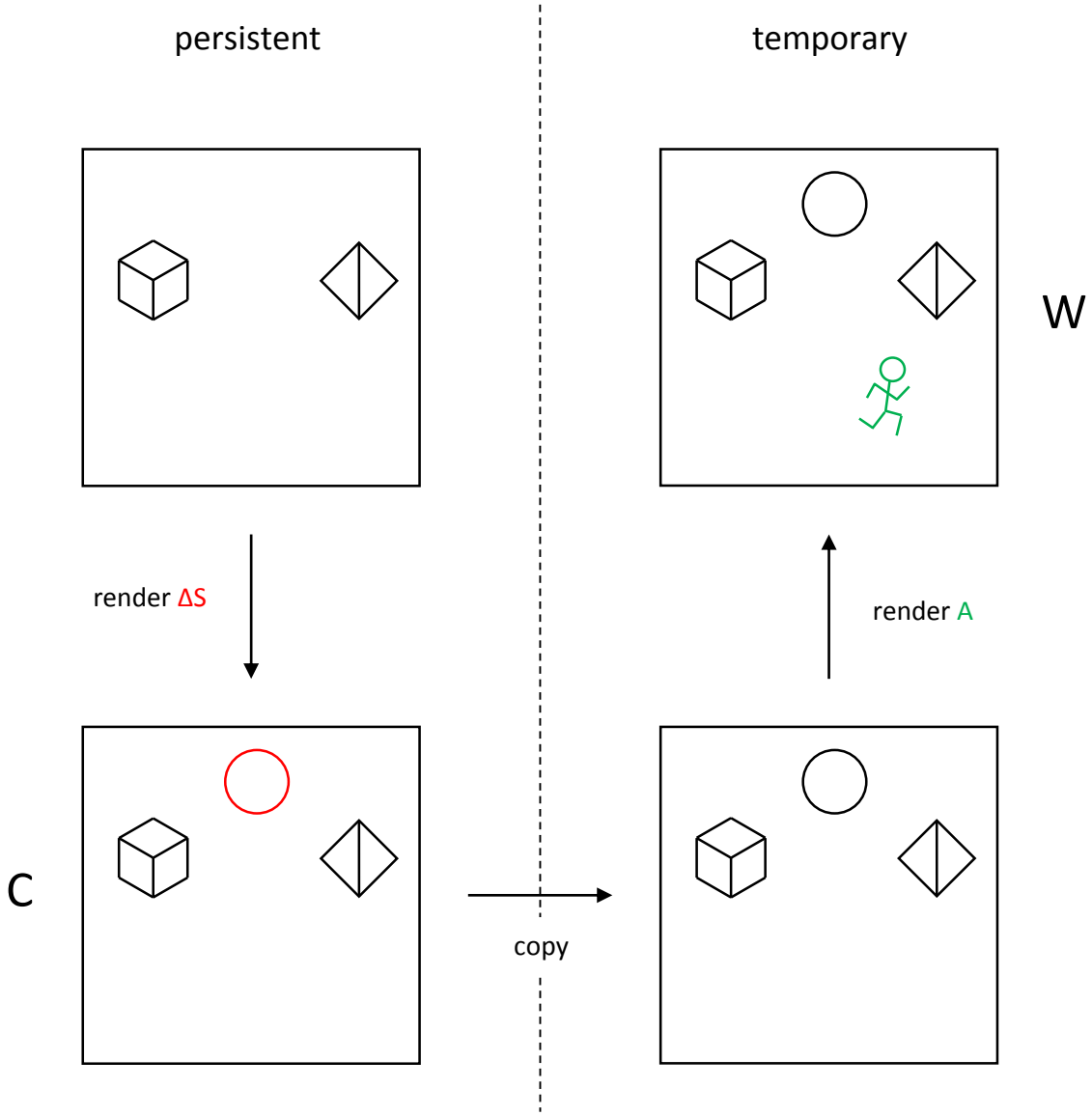

Figure 1: *shadow map caching scheme*

3. CSM scrolling

When dealing with the common case of local light sources, the shadow map caching technique is generally only usefully when the shadow projection has remained stationary since the previous frame. On the other hand, in the special case of cascaded shadow maps for a constant sunlight direction, the projection changes frequently as the camera moves and rotates but it does so in a particularly simple way. Consider first just a translation of the camera: if movement is perpendicular to the light direction, the 2D map contents are scrolled left-right and/or up-down. If the movement is parallel to the light direction, the contents of the map are translated in depth – that is to say, each pixel of the map is incremented or decremented by the amount through which the camera has moved. A general translation of the camera will involve some combination of these two effects.

While a rotation of the camera can often bring about larger movements of the CSM view volumes, these are still translational movements and so the contents of the maps will be affected in the same kind of way, albeit the magnitude of the movement may be larger.

These observations lead to the notion that the shadow map caching scheme described earlier, when combined with a technique for scrolling portions of map data, can be extended to cover cascaded shadow maps. The basic principle is to scroll the static portion of the map which is common to both the current frame and the previous one, and then fill in the gap at the edge with newly rendered static geometry. Ignoring for the moment the low-level specifics of how the scrolling takes place, the CSM version is shown in the pseudocode below, and figure 2 summarizes the mechanism outlined in the pseudocode.

Note: when retrieving S', the list of static objects from the previous frame's query, it is important to use a cached version of their positional information from the previous frame. The current frame's information about scene objects is insufficient here because some previously static objects may have been moved or removed.

Crucial to the algorithm's performance is to use a description of the scene in which geometry is broken into relatively compact chunks. Nothing will be gained by rendering the scene as a single monolithic element, since the full rendering cost would be incurred even for the smallest of camera movements.

From figure 2 it becomes apparent that the set ΔS comprises both newly static objects, and previously static objects which have scrolled into view since the last frame.

```
// to generate working CSM map W, having query volume V,
// while maintaining cached CSM map C

invalidate C
S={}

for each frame
{
  let V'=V
  update V

  if projection changed in any way other than translation
    invalidate C

  for each scene object
    flag as static or active

  let S'=S

  query scene to find:
  S = {static objects intersecting V and not occluded}
  A = {active objects intersecting V and not occluded}

  if C is valid
  {
    if any member of S' is no longer static or has been deleted
    {
      let ∩V = V∩V'
      query S' to find ∩S = {members of S' intersecting ∩V}
      if any member of ∩S is no longer static or has been deleted
        invalidate C
    }
  }

  if C is valid
  {
    // reuse cached map
    if V!=V'
      scroll C
    let ΔS = S-S'
    render ΔS to C
  }
  else
  {
    // redraw cached map
    clear C
    render S to C
  }

  flag C as valid
  copy C to W
  render A to W
}
```

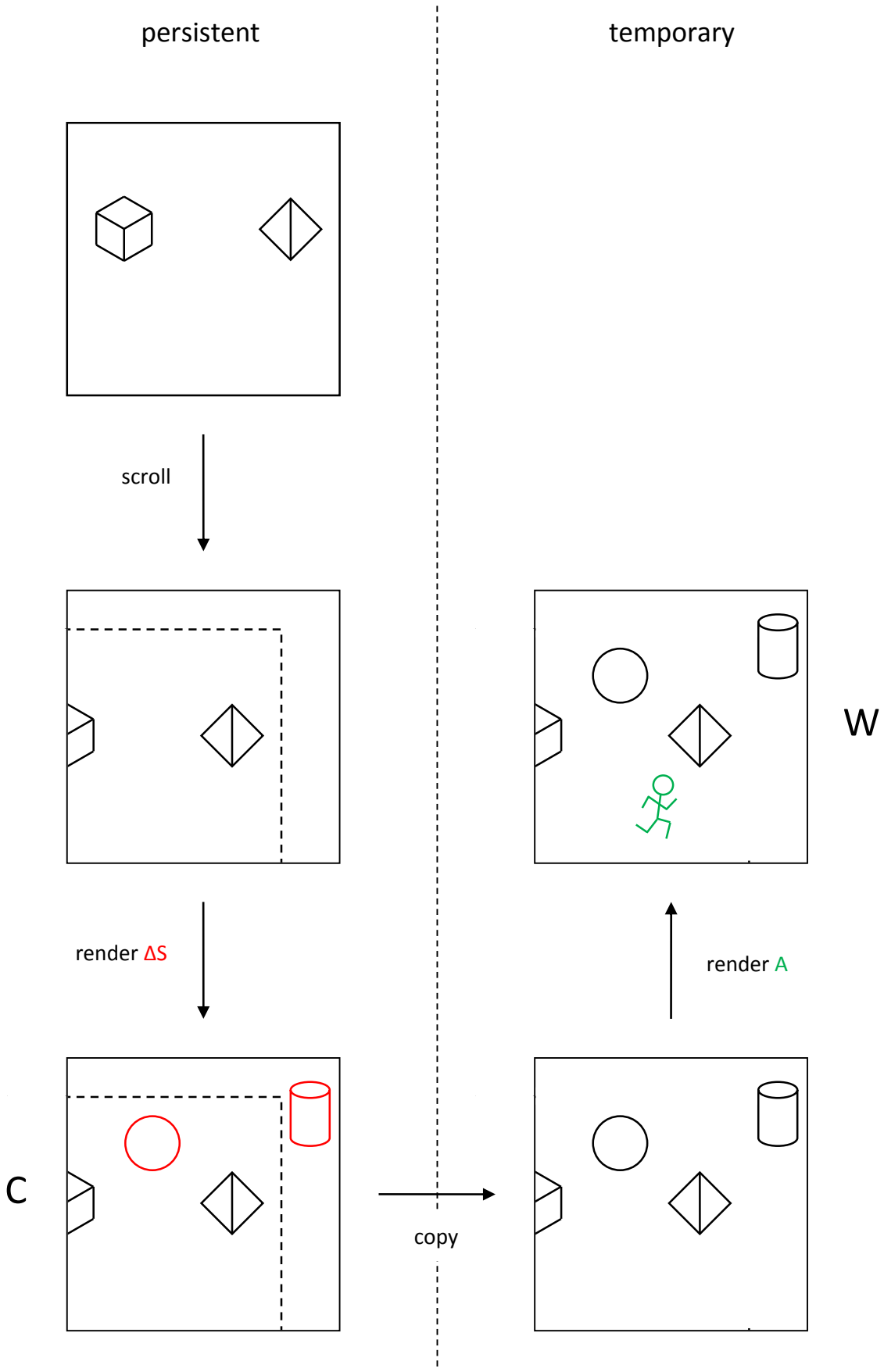

Figure 2: *CSM scrolling mechanism*

4. Low-level scrolling mechanism

This is an operation well suited to a gpu, since it is not substantially different from simple texture mapping with a constant uv-offset at each pixel.

4.1 Lateral scrolling

First, suppose that the camera movement is a translation perpendicular to direction of the light rays. No depth changes will be incurred, and a simple uv-translation of the existing texture data suffices. This can be performed using ordinary texture mapping, expressed in a pixel shader as follows:

```
float ScrolledDepth_LateralOnly( input )
{
  float2 uv = input.xy;
  return SampleShadowMap(uv);
}
```

The texture lookup here uses point-sampling. The vertex shader simply supplies the interpolated texture coordinates, and a full-screen polygon is drawn with suitably offset texture coordinates. The computation of the offset is described in section 5.

4.2 Scrolling in the depth dimension

To achieve a movement parallel to the light direction, we must add a depth offset value to each sampled depth. A general translation along all three axes is therefore obtained by ordinary texture mapping together with an added depth offset:

```
float ScrolledDepth( input )
{
  float2 uv = input.xy;
  float  depth_offset = input.z;
  float  old_depth = SampleShadowMap(uv);
  return old_depth + depth_offset;
}
```

The third channel of the input interpolator has been used to pass the depth offset, which is constant over the whole buffer.

4.3 Treatment of the boundaries

Any attempt to read a texel beyond the boundary of the map should be treated as an unknown depth to be subsequently filled in with newly rendered data. That is to say, it should be cleared to the depth buffer clear value, 1.0. This is most easily achieved by setting clamp-to-border mode together with a border colour of 1.0.

Equally important is the treatment of depths lying outside the near-far range. The hardware will usually clamp the result to the [0.0, 1.0] range before writing it out, as values beyond this range cannot be represented in the map. We must consider whether this clamping behaviour is desirable.

If depth-offsetting causes a pixel's new depth to lie on the near side of the near plane, i.e. a negative depth value, it is acceptable to clamp the depth to 0.0. While the results will differ from those obtained by redrawing from scratch, the differences are generally unimportant. Near-clipping of geometry rendered into a shadow map is always undesirable, with or without the scrolling technique, and the rendering volume will usually have been extended far enough towards the light to encompass all relevant shadow casters. Thus the difference will not generally arise in practice. Indeed the non-standard behaviour of the scrolled map version can arguably be seen as more desirable, because to some degree it can prevent near-clipping from cutting holes in shadows from distant casters.

The situation at the far plane is a little different. While clamping out-of-range values to 1.0 is sensible, an additional treatment is necessary. Suppose the *looked-up* depth is already 1.0, corresponding to the far plane, and suppose also that the depth offset is negative. If we simply added the depth offset and wrote out the resulting depth, it would now represent a surface closer than the far plane. The problem with this naïve approach is that 1.0 is also the clear-value of the depth buffer – i.e. a point lying exactly on the far plane is indistinguishable from a pixel which simply hasn't been written to since the buffer-clear. We certainly don't want the depth-offsetting to pull the entire back wall of the render volume forward in depth. The sensible course of action therefore is to assume that depths of 1.0 represent unwritten pixels, and to leave them at 1.0. In the absence of a standard hardware setting to support this operation, we add a conditional test in the pixel shader. The final version is:

```
float ScrolledDepth( input )
{
  float2 uv = input.xy;
  float  depth_offset = input.z;
  float  old_depth = SampleShadowMap(uv);
  float  new_depth = old_depth + depth_offset;
  return (old_depth < 1.0) ? new_depth : 1.0;
}
```

5. Managing the render volume associated with a map

A square texture resolution (512x512 in our system) is chosen upfront to apply to all maps. The portion of the camera frustum requiring shadow mapping is split into (typically 4) sub-frustums by splitting planes located at chosen depths along the camera's z-axis, as in the standard CSM algorithm presented in [Engel 06].

Now a render volume is assigned to each map. The physical size of the volume must be large enough to encompass the diameter (furthest distance between any two points) of the corresponding camera sub-frustum, expanded by one texel in each direction to permit the whole-texel movement decribed next. Note that it is unnecessary to encompass the bounding sphere of the sub-frustum, contrary to some authors' claims. We make use of the entire area of each map for rendering geometry, regardless of whether it lies outside the 'shadow' of the camera sub-frustum.

The render volume should only be moved by whole-texel amounts in the two axes perpendicular to the direction of light rays, in order to maintain the stability of stationary shadows when the camera is in motion, as described in [Valient 08]. With CSM scrolling there is an additional reason for wanting this constraint: to maintain the integrity of the scrolled shadow data. In this case it is also desirable to

constrain the movement *parallel* to the light rays, so that the render volume shifts by whole depth-units, e.g. 2^{-16} in the case of a 16-bit depth buffer.

In our system, to determine the location of the render volume for a given map, we first transform the camera sub-frustum into the light's coordinate frame – an orthogonal frame with z-direction parallel to the light rays and the x- and y-axes constrained in the usual way by the world-up direction, and with the origin made to coincide with the world origin. We then find the axis-aligned bounding box of the sub-frustum, in light-space coordinates, and use this to position the centre of the render volume. Finally we round the coordinates to whole texel units and whole depth units.

The resulting coordinates are used in 2 ways: by transforming them back to world coordinates we can position the query volume for culling and the render volume for rendering; and by maintaining a 1-frame history of the light-space coordinates we have all the information needed to generate the scroll amount for the current frame. The following code indicates how the position of the AABB in light-space coordinates is converted into the vertex-level values passed into the shaders in section 4:

```
texel_coords = VecFloor(aabb_centre * meters_to_texels);

du = (texel_coords.x - prev_texel_coords.x) / texture_width;
dv = (texel_coords.y - prev_texel_coords.y) / texture_height;
dw = (texel_coords.z - prev_texel_coords.z) / depth_range;

x0 = 0.0f;
y0 = 0.0f;
x1 = tex_width;
y1 = tex_height;
u0 = 0.0f + du;
v0 = 0.0f + dv;
u1 = 1.0f + du;
v1 = 1.0f + dv;

verts[0].Set(x0,y0,u0,v0,-dw);
verts[1].Set(x0,y1,u0,v1,-dw);
verts[2].Set(x1,y0,u0,v0,-dw);
verts[3].Set(x1,y1,u1,v1,-dw);

DrawQuad(verts);
```

Side note: one possible technique for reducing the amount of time spent rendering to the map is to provide a guard band around the map proper, a region which redundantly stores shadow map data despite lying outside the parts of the map sampled when rendering to the frame buffer. The benefit of such an 'extended map' is that we can allow the region corresponding to the map proper to shift around inside the extended map, without the need to scroll or to redraw static geometry until the map proper needs to move outside the guard band. There is a trade-off here, of course – the reduced average rendering time comes at a cost of either larger maps or coarser shadows.

6. Results

In figure 3, the shadow depth buffers are shown for the four levels of the CSM under a translational movement of the camera. The grey buffers show the result of unconditional rendering of all geometry, the blue buffers show the portion of cached data immediately following the scrolling operation, and the red buffers show the newly visible static geometry to be rendered over the top of the the scrolled data. No active scene objects are used in this example.

Note the white strips at the bottom and right of the blue maps; this is the region set to the buffer's clear-colour (1.0) by the scrolling process and represents the area to be filled by rendering the set ΔS. A certain amount of scrolling in the depth direction is occurring too, but the render volumes are long enough that they don't cut any geometry at their near or far planes.

The speed of camera motion in figure 3 is about 15 ms^{-1} (at 30 fps). Notice that the size in texels of the blank strips in the blue maps varies in inverse proportion to the physical size of the map.

Figure 4 shows the corresponding results for a rotational camera motion of about 210 degrees per second (at 30 fps). In this case, the blank strips are roughly the same size in texels for all 4 maps, because each camera sub-frustum sweeps round at a velocity proportional to its distance from the viewpoint.

The magnitude of both of these movements is on the higher end of what might be encountered in a typical in-game scenario. The relative absence of red geometry, particularly in the coarser maps toward the right side, demonstrates the value of the technique.

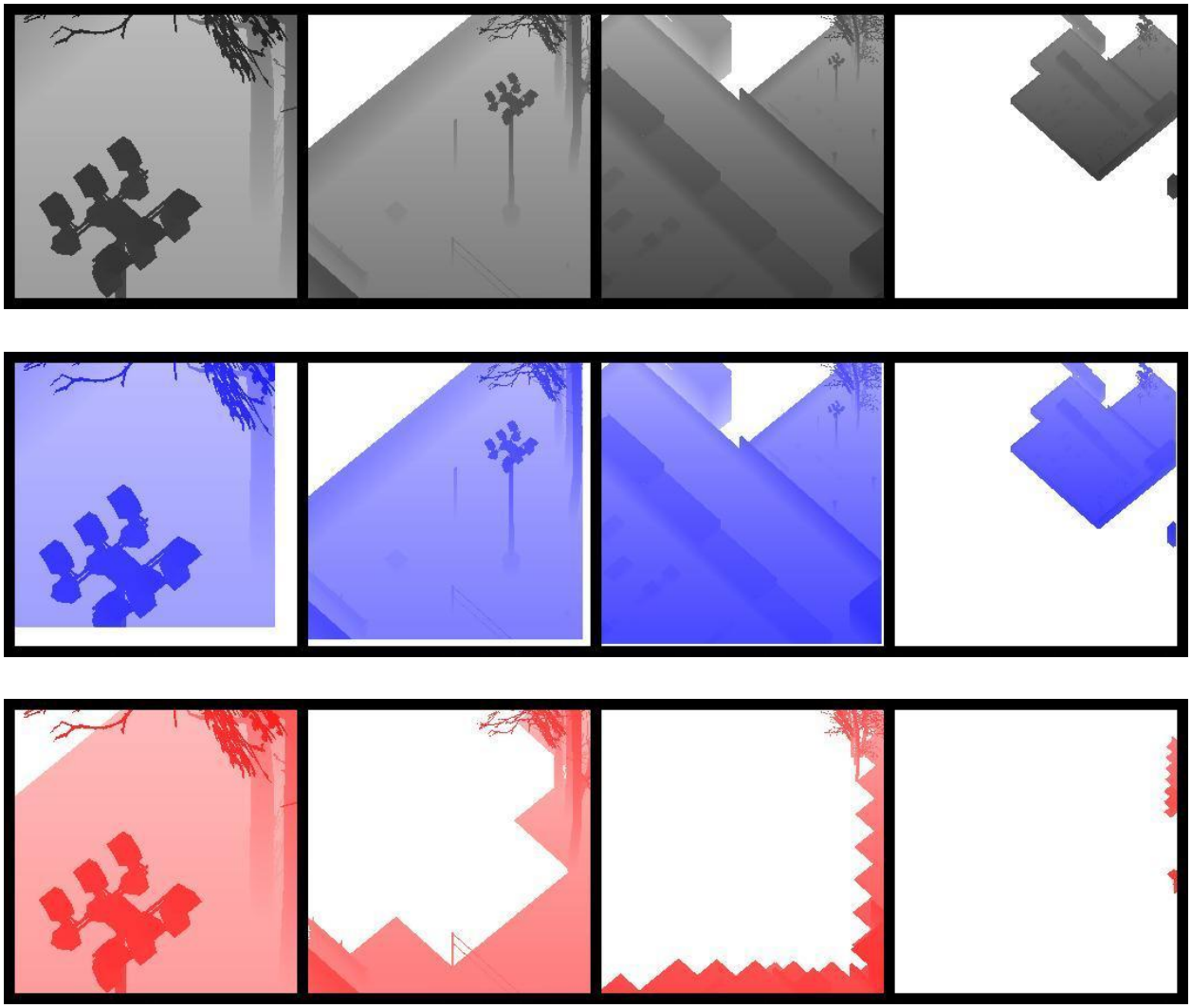

Figure 3: *depth buffers of the 4 levels of detail under camera translation. From top: the fully rendered buffers for reference; the cached data scrolled from the previous frame; the newly scrolled-in geometry to be rendered onto the scrolled maps.*

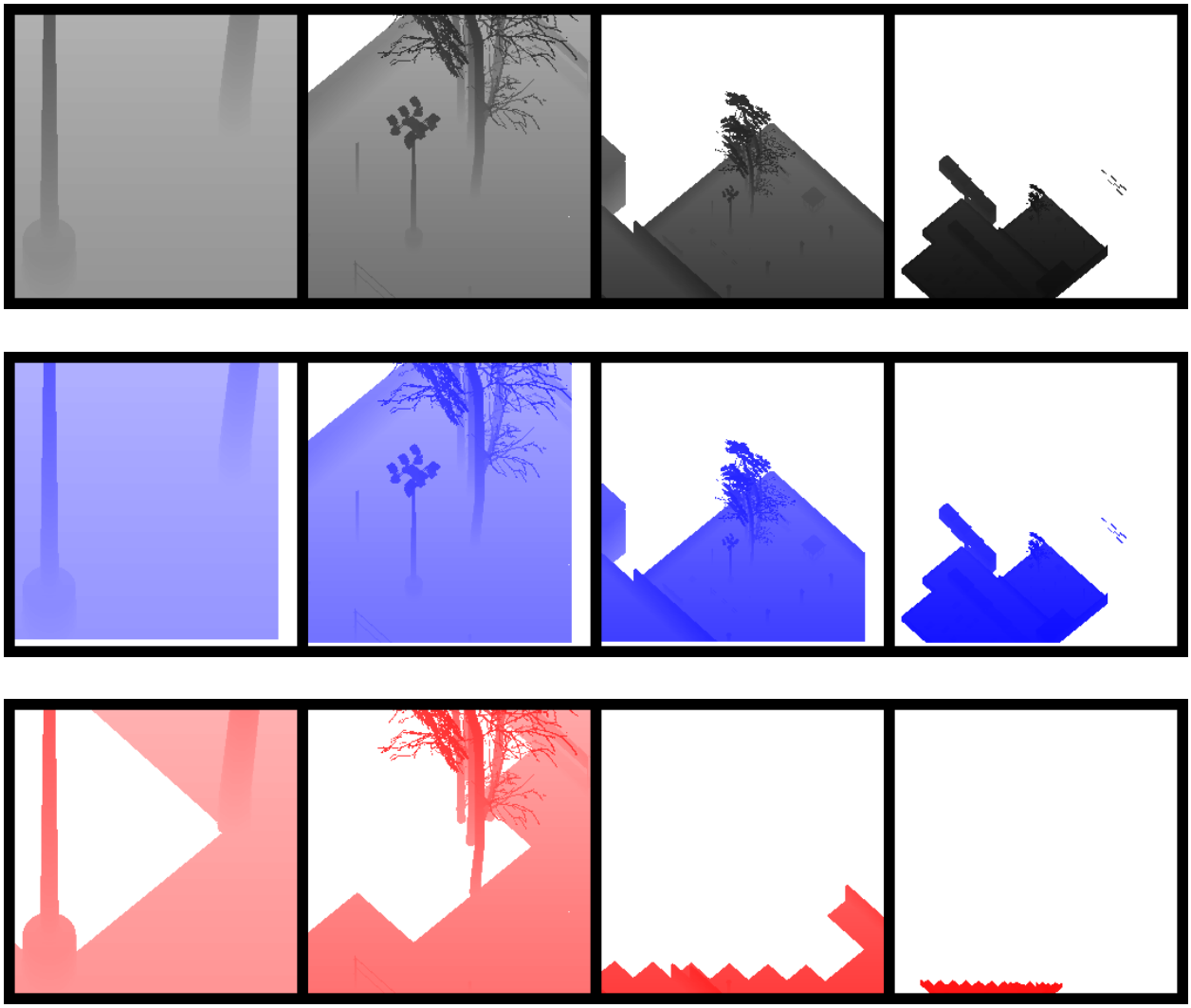

Figure 4: *depth buffers of the 4 levels of detail under camera rotation.*

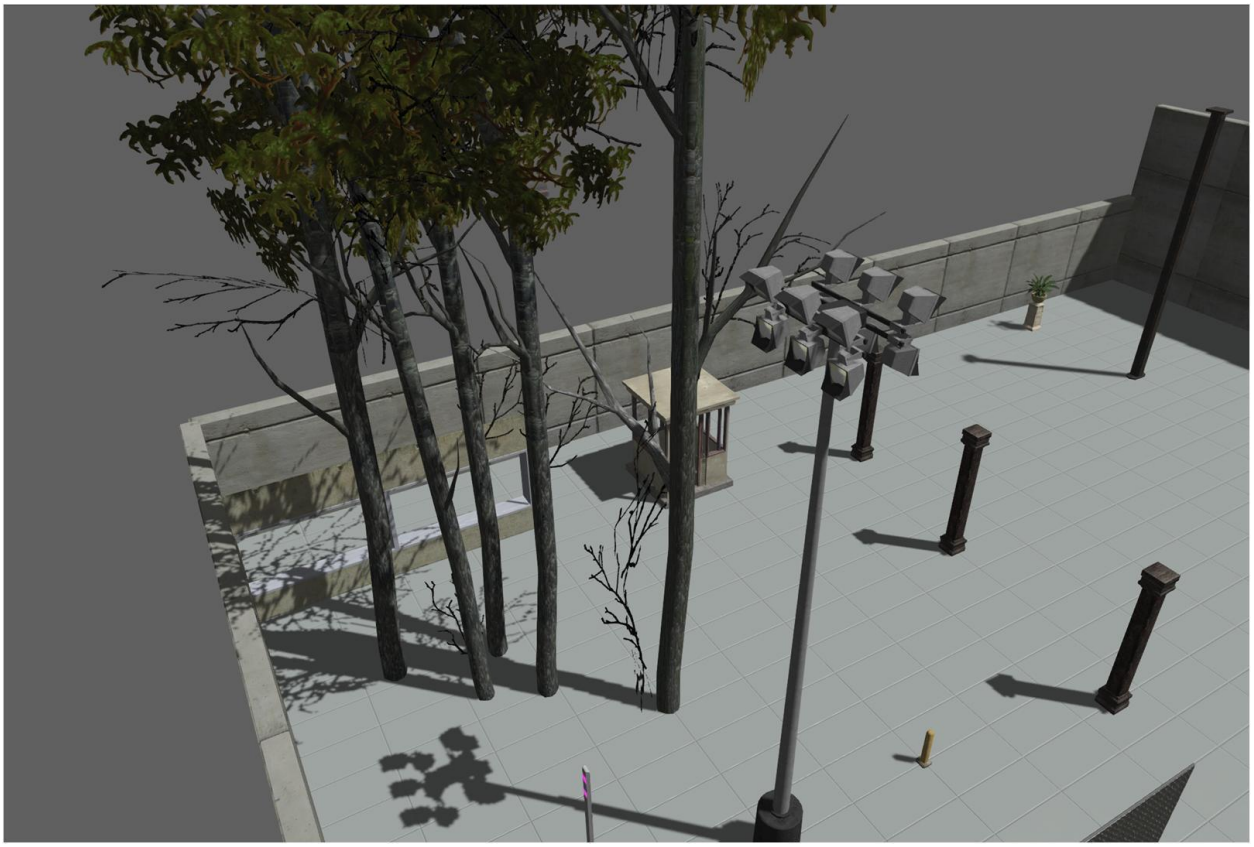

Figure 5: *for illustration, a rendering of the scene being used.*

7. Bibliography

[Engel 06] W. Engel. “Cascaded shadow maps”, in: Engel, W.F., et al., “ShaderX5 – Advanced Rendering Techniques”, Charles River Media, 2006.
[Valient 08] M. Valient. “Stable rendering of cascaded shadow maps”, in: Engel, W. F., et al., “ShaderX6 – Advanced Rendering Techniques”, Charles River Media, 2008.