Shadow maps and projective texturing in X3D

Michalis Kamburelis*
Institute of Computer Science
University of Wrocław, Poland

Abstract

We propose a number of X3D extensions to enable shadows in the virtual worlds. Our higher-level extensions are an easy way to request shadows independently of their implementation. Lower-level extensions allow to control the details of *shadow maps* generation and *projective texture mapping*. Together, they allow the authors to activate real-time dynamic shadows on 3D scenes. The extensions expose also projective texture mapping for purposes other than shadows, for example we can cast a color texture from a light source. Introduced concepts map naturally to any basic shadow maps implementation, and integrate nicely with existing X3D components like textures and shaders.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages, Standards

Keywords: X3D graphics, shadows, shadow maps, projective texturing

1 Introduction

X3D [Web 3D Consortium 2008] is an open standard for representing rich 3D data. Many advanced graphic effects are available for the creators of interactive 3D worlds.

Shadow maps [Williams 1978] are one of the major approaches for generating real-time dynamic shadows. They are relatively simple to implement, supported by graphics hardware (both in fixed-function pipeline and shaders), and with proper implementation can achieve very good quality. They work with any geometry, including difficult cases like 3D geometry that is not correctly closed, flat 2D geometry and alpha-test textures. They have no problems with dynamic environments.

A closely related subject is *projective texturing* [Everitt 2001a]. It is utilized by the shadow maps algorithm, but is useful also in other circumstances, when you want to „cast" a texture from a light.

In this paper we introduce a number of X3D extensions to use and control shadow mapping and projective texturing. Our extensions allow the authors of virtual worlds to easily use shadow maps in the scene. They also expose the most useful shadow mapping parameters, so that they can be tuned to achieve the best results. Authors also get control over projective texturing, that may be used with any kind of textures — like depth textures generated by shadow maps, the standard color textures, and any other texture types expressible in X3D.

Authors that attach shaders to the geometry (for example using the OpenGL Shading Language, GLSL) get explicit control over what happens with the shadow map values. This allows to use our shadow maps in non-trivial scenarios, combining them with a custom shading methods and such. Also *percentage closer filtering* [Bunnell and Pellacini 2004] can be trivially implemented inside the shader.

For the developers of X3D browsers, our extensions strive to be relatively easy to implement. Any basic shadow maps implementation should naturally map to presented new nodes and fields. An example open-source implementation, supporting all the extensions' features, is available inside the VRML/X3D engine on `http://vrmlengine.sourceforge.net/`.

2 Shadow maps algorithm

For the purpose of better understanding the following extensions, we present here a short overview how shadow mapping works. This is intentionally a simplified overview, implementors will most definitely want to look at more detailed articles like NVidia presentations [Everitt 2001b], [Everitt et al. 2001]. Shadow maps are suitable for implementation in both OpenGL, Direct3D, and in software renderers.

*e-mail: michalis.kambi@gmail.com

The first step is to **generate the shadow map texture**. We place the camera at the position of a light source, and point it in the light's direction. Then we render the depth buffer of the scene to a *depth texture*. This step must be done before rendering the actual scene with shadows.

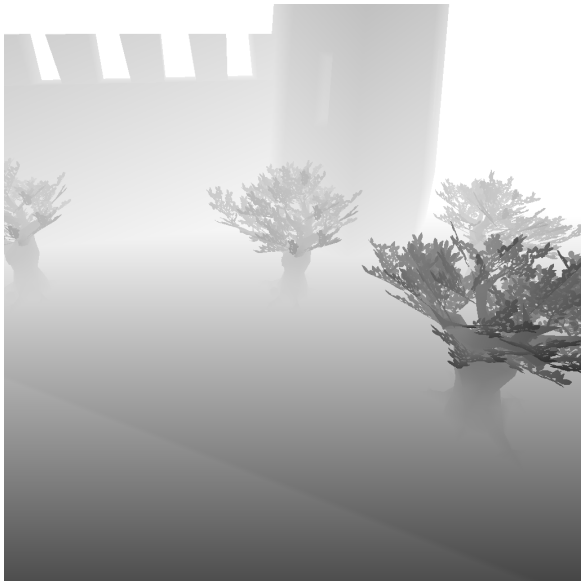

Figure 1: *Shadow map, as seen from the light source. Darker colors mean that the object is closer to the light source.*

Since a point light source doesn't have a direction, a common solution is to render *six* depth textures along the six major directions (-/+X, -/+Y, -/+Z, usually in the global coordinate space). This way we map the view around an imagined cube around the light source.

For a directional light source, we need to know its position. Although conceptually directional light is positioned at infinity, for shadow maps we must assign a normal 3D position even to a directional light. Together with the projection near and far plane, this position is needed to determine what depths will be captured accurately, that is which shadow casters will cast a correct shadow.

For light sources that have parallel rays (like a directional light), we use orthographic projection. For others (like a positional or spot light) we use perspective projection.

Once we have a shadow map, the second step is to **map the generated texture on shadow receiving geometry**. This is done during the normal rendering of the scene, when the camera corresponds to the avatar view. We want to map the texture treating the light source like a projector that „casts" the texture onto the scene. This stage of the algorithm is known as the *projective texturing*, and can be used as well for other purposes, like projecting standard color textures over the scene.

Knowing the lights projection parameters and the current camera parameters we want to calculate the texture coordinate t for a point p like

$$t = S * L_p * L_v * C_v^{-1} * p \tag{1}$$

Where L_p is the light projection matrix, L_v is the light view matrix, and C_v is the camera (representing the avatar) view matrix. S is a trivial constant matrix to scale and shift clip coordinates by $\frac{1}{2}$, to have the texture coordinates in $[0..1]$ range.

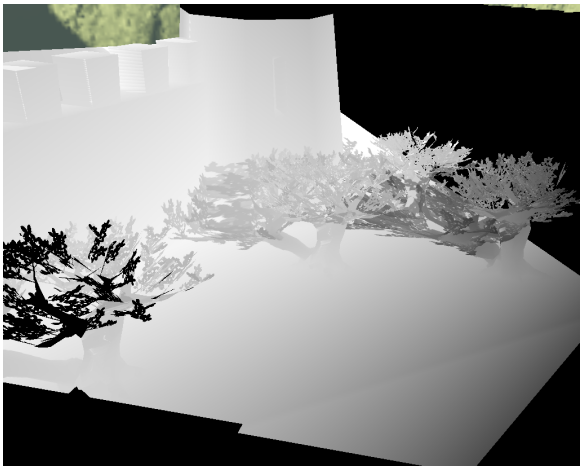

Figure 2: *Shadow map, as seen from the avatar camera. This is the previous shadow map projected over the scene.*

Calculated $(t.x, t.y)$ can be used to sample the shadow map, to get the distance from the light to the object that obscures the light source at this point. Calculated $t.z$ contains the distance from the light to the current point. Thus the third and final step is to **determine if the point lies in the shadow** by comparing

$$t.z > texture(shadowMap, t.x, t.y) \tag{2}$$

Note that in 3D graphics we use 4x4 matrices, and 3D positions are expressed in the 4D homogeneous coordinates. The vectors after perspective transformation will have the 4th component $(t.w)$ different than 1, so this cannot be ignored. So the real equation is actually

$$\frac{t.z}{t.w} > texture(shadowMap, \frac{t.x}{t.w}, \frac{t.y}{t.w}) \tag{3}$$

For simplicity, let's call $t.z/t.w$ simply a d (for *distance to the point*).

When the object is not in the shadow, d is equal to the texture value (we're looking at the object obscuring the light). Otherwise, it's larger (we're looking at the object *behind* the one obscuring the light). In a perfect world (assuming an infinite resolution of the shadow map, no floating point errors etc.) d should never be smaller than the right side of the equation.

Careful reader will notice now that the world is not actually perfect. Precision of the depth values is limited, and gets worse the farther away from the light source we are. The shadow map resolution never matches perfectly the screen resolution (one shadow map pixel may correspond to many screen pixels). And finally we have to account for the floating point calculation errors. Due to these problems, a small offset is needed to make the comparison with d above behave stable. Our extensions encourage the implementation to apply this offset when generating the shadow map, and allow the author to adjust the offset parameters for particular cases.

At the end we want to actually use the results of the above comparison. That is, we want to **render the places in shadow with darker colors**. In the simplest case, we can simply make the surface black when it is in the shadow or use the original color when surface is lit. This step is controlled fully in the shader, so shader authors may apply here any shading algorithm they see fit.

3 Overview and discussion

Many scene elements cooperate to produce a nice shadow effect: light sources, shadow casters and shadow receivers. This enables various potential ways for spreading the shadow information over the X3D nodes. Our extensions put the main weight on the *shadow receivers*. You have to explicitly designate shapes receiving the shadow: either by the simple `Appearance.receiveShadows` field, or by using the lower-level tools `GeneratedShadowMap`, `ProjectedTextureCoordinate` and custom shaders.

Specifying additional shadow information at the light sources and shadow casters is purely optional. Every light source may cast shadows, and everything is a shadow caster by default.

Compare this with other approaches by BS Contact [Bitmanagement 7.2], [Bitmanagement 6.2] and Octaga [Octaga]. Their approaches favor specifying shadow information in separate nodes (`Shadow` in Octaga, `ShadowGroup` in BS Contact) that explicitly enumerate shadow casters (occluders, emitters) and receivers.

Both Octaga and BS Contact provide also lower-level nodes for storing the shadow maps (`ShadowTexture` in Octaga, for BS Contact this is a special case of `CompositeTexture3D`). This is analogous to our `GeneratedShadowMap` node and serves a similar purpose: author can get more control over shadow mapping this way. In particular, custom fragment shaders may be used to visualize shadows.

Various needs guided the design of our shadow extensions:

1. **Flexibility of what casts and receives the shadow**. In the extreme case, we could specify on each shadow receiver from which lights it receives shadow, and for every combination of light and shadow receiver — which shapes occlude the light. We think this is too burdensome for typical uses, and stands in the direct opposition to the goal of **easily enabling shadows on the existing scenes**.

   Our extensions allow to choose only lights on the shadow receivers. Configuration of shadow casters is global, that is an object either casts a shadow on all the shadow receivers, or doesn't cast a shadow at all.

2. **Flexibility of what shadow method is used**. Our extensions, as well as BS Contact and Octaga, are strongly directed at the shadow mapping algorithms. Our extensions are also usable for other shadow approaches, as long as the author uses only the most encouraged `Appearance.receiveShadows` and the `Appearance.shadowCaster` extensions.

   The important point here is that our `receiveShadows` field can work really well, so many scenes are fine using it and thus work with any shadow algorithm.

3. **Natural behavior for authors**. Not much technical knowledge should be required from the authors, and the shadow properties should be declared where they feel most natural. We think that our choice of extending the shadow receivers is the most natural here — receiving the shadow changes the look of given shape.

   We also note that this nicely fits with **natural placement for implementors**. In case of shadow maps, receiving the shadows requires shading the object differently.

Now that we summarized what we want, let's quickly review where we can place shadow information in the X3D scene:

1. Completely **separate** nodes, like a special `Shadow` node, do not encourage authors to use shadows widely.

2. Requiring much configuration of the **shadow casters** seems unnecessary, because most shadows algorithms easily account for a large number of shadow casters. This is true for shadow maps, as well as shadow volumes and ray testing. The only exception could be plane-projected shadows, but these depend heavily on the number of shadow receivers as well.

   So the best treatment of shadow casters is to make everything, by default, a shadow caster, and do not require any configuration for them. Algorithms that are internally unable to cope with some shadow casters can simply ignore them. For example, shadow volumes implementations using classic silhouette optimization may want to detect and ignore shapes that are not 2-manifold.

3. Placing information at **light sources** is natural, and our extensions put some information there.

   For shadow algorithms that, in their basic implementation, make everything a shadow receiver (like multi-pass shadow volumes, or shadow ray testing), it would be possible to make everything a shadow receiver by default (and thus, do not require any shadow receivers configuration). This could be a working solution for small, simple scenes. But for large scenes, we need more flexibility to limit the shadows.

4. Thus, the explicit configuration of **shadow receivers** seems like a best choice for us. It is flexible, and still simple enough to be widely used on many shapes in the scene. Authors just add lights to the `Appearance.receiveShadows` field in the simplest case.

   Moreover, receiving shadows changes the look of the given shape. This is a very practical insight, because it means that usage of shaders on shadow receivers is limited. For shadow maps, shadow receiver must use the appropriate shader. When using the `receiveShadows` field, author must be aware that browser may force usage of internal shaders. When placing `GeneratedShadowMap` on the `textures` list the author must be aware that he must write his own shader to get really nice shading results.

4 X3D extensions

We present now the actual X3D extensions to enable shadow maps and projective texturing. We propose to add a couple of new fields to the existing nodes, and some new nodes.

The specification of nodes and fields in this section is similar to the X3D specification conventions:

```
FieldType [in,out] fieldName <default value>
  # optional range of allowed values
  # for the field above
```

4.1 Define shadow receivers

In the simplest case, to enable the shadows authors must only use this field:

New field for the Appearance node

```
MFNode  []  receiveShadows  []
  # [X3DLightNode] list
```

Each light present in the `receiveShadows` list will cast shadows on the given shape. That is, contribution of the light source will be scaled down if the light is occluded at a given fragment. The whole light contribution is affected, including the ambient term. We do not make any additional changes to the X3D lighting model. The resulting fragment color is the sum of all the visible lights (visible because they are not occluded, or because they don't cast shadows on this shape), modified by the material emissive color and fog, following the X3D specification.

This is the simplest extension to enable shadows. It is suitable for any shadows implementation, not only shadow maps.

Authors should note that browsers may use internal shaders to produce nice shading for shadow receivers. Custom author shaders may be ignored. If you want to apply your own shaders over shadow receivers, you have to use the lower-level nodes described next instead of this.

4.2 Overview of the lower-level extensions

The following extensions make it possible to precisely setup and control shadow maps. Their use requires a basic knowledge of the shadow map approach, and they are necessarily closely tied to the shadow map workflow. On the other hand, they allow the author to define custom shaders for the scene and control every important detail of the shadow mapping process.

These lower-level extensions give a complete and flexible system to control the shadow maps, making the `receiveShadows` feature only a shortcut for the simplest setup.

We make a shadow map texture by the `GeneratedShadowMap` node, and project it on the shadow receiver by `ProjectedTextureCoordinate`. An example X3D code (in classic encoding) for a shadow map setup:

```
DEF MySpot SpotLight {
  location 0 0 10
  direction 0 0 -1
  projectionNear 1
  projectionFar 20
}

Shape {
  appearance Appearance {
    texture GeneratedShadowMap {
      light USE MySpot
      update "ALWAYS"
    }
  }
  geometry IndexedFaceSet {
    texCoord ProjectedTextureCoordinate {
      projector USE MySpot
    }
    # ... other IndexedFaceSet fields
  }
}
```

4.3 Light sources parameters

The motivation behind the extensions in this section is that we want to use light sources as cameras. This means that lights need additional parameters to specify projection details.

To every X3D light node (`DirectionalLight`, `SpotLight`, `PointLight`) we add new fields:

Additional fields for every X3DLightNode

```
SFFloat  [in,out]  projectionNear 0
  # must be >= 0
SFFloat  [in,out]  projectionFar 0
  # must be > projectionNear, or = 0
SFVec3f  [in,out]  up 0 0 0
SFNode   []        defaultShadowMap  NULL
  # [GeneratedShadowMap]
```

The fields `projectionNear` and `projectionFar` specify the near and far values for the projection used when rendering to the shadow map texture. These are distances from the light position, along the light direction. You should always try to make `projectionNear` as large as possible and `projectionFar` as small as possible, this will make depth precision better (keeping `projectionNear` large is more important for this). At the same time, your projection range must include all your shadow casters.

The field `up` is the „up" vector of the light camera when capturing the shadow map. This is used only with non-point lights (`DirectionalLight` and `SpotLight`). Although we know the direction of the light source, but for shadow mapping we also need to know the „up" vector to have camera parameters fully determined. This vector must be adjusted by the implementation to be perfectly orthogonal to the light direction, this allows user to avoid explicitly giving this vector in many cases. Results are undefined only if this vector is (almost) parallel to the light direction.

These properties are specified at the light node, because both shadow map generation and texture coordinate calculation must know them, and use the same values (otherwise results would not be of much use).

The field `defaultShadowMap` is meaningful only when some shape uses the `receiveShadows` feature. This will be described in the later section 4.7.

`DirectionalLight` gets additional fields to specify orthogonal projection rectangle (projection XY sizes) and location for the light camera. Although directional light is conceptually at infinity and doesn't have a location, but for making a texture projection we actually need to define the light's location.

Additional fields for the DirectionalLight node

```
SFVec4f  [in,out]  projectionRectangle
  0 0 0 0 # left, bottom, right, top;
  # must be left < right and bottom < top,
  # or all zero
SFVec3f  [in,out]  projectionLocation  0 0 0
  # affected by node's transformation
```

`SpotLight` gets additional field to explicitly specify a perspective projection angle.

Additional fields for the SpotLight node

```
SFFloat  [in,out]  projectionAngle  0
```

Leaving `projectionAngle` at the default zero value is equivalent to setting `projectionAngle` to `2 * cutOffAngle`. This is usually exactly what is needed. Note that the `projectionAngle` is the vertical and horizontal field of view for the square texture, while `cutOffAngle` is the angle of the

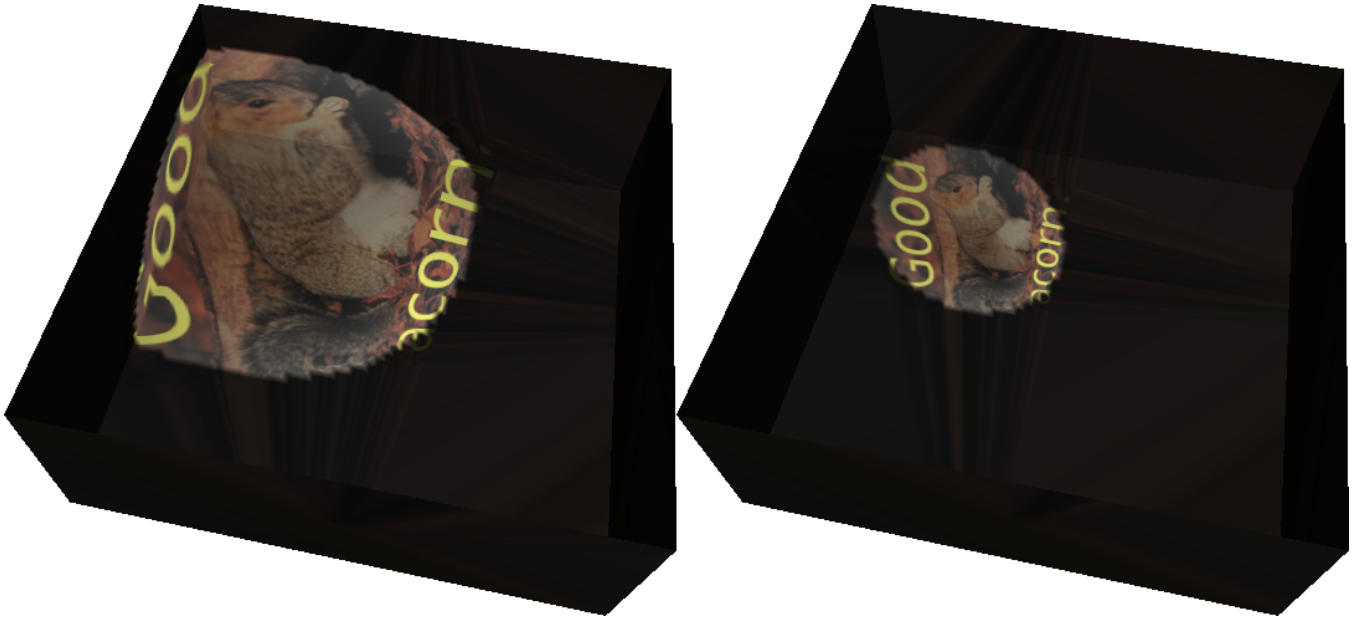

Figure 3: *Spot light cone angle matching with the projected texture angle*

half of the cone (that's the reasoning for $*2$ multiplier). Using `2 * cutOffAngle` as `projectionAngle` makes the perceived light cone fit nicely inside the projected texture rectangle. It also means that some texture space is essentially wasted — we cannot perfectly fit a rectangular texture into a circle shape.

Figure 3 shows how a light cone fits within the projected texture.

4.4 Automatically generated shadow maps

Now that we can treat lights as cameras, we want to render shadow maps from the light sources. The rendered image is stored as a texture, represented by a new node:

GeneratedShadowMap : X3DTextureNode

```
SFNode    [in,out]  metadata      NULL
  # [X3DMetadataObject]
SFString  [in,out]  update        "NONE"
  # ["NONE" | "ALWAYS" | "NEXT_FRAME_ONLY"]
SFInt32   []        size          128
SFNode    []        light         NULL
  # [X3DLightNode] (any light node) allowed
SFFloat   [in,out]  scale         1.1
SFFloat   [in,out]  bias          4.0
SFString  []        compareMode
  "COMPARE_R_LEQUAL" # ["COMPARE_R_LEQUAL"
  # | "COMPARE_R_GEQUAL" | "NONE"]
```

The `update` field determines how often the shadow map should be regenerated. It is analogous to the `update` field in the standard `GeneratedCubeMapTexture` node.

`"NONE"` means that the texture is not generated. It is the default value (because it's the most conservative, so it's the safest value).

`"ALWAYS"` means that the shadow map must be always accurate. Generally, it needs to be generated every time shadow caster's geometry noticeably changes. The simplest implementation may just render the shadow map at every frame.

`"NEXT_FRAME_ONLY"` says to update the shadow map at the next frame, and afterwards change the value back to `"NONE"`. This gives the author an explicit control over when the texture is regenerated, for example by sending `"NEXT_FRAME_ONLY"` values by a `Script` node.

The field `size` gives the size of the (square) shadow map texture in pixels.

The field `light` specifies the light node from which to generate the map. Ideally, implementation should support all three X3D light source types. `NULL` will prevent the texture from generating. It's usually comfortable to `"USE"` here some existing light node, instead of defining a new one.

Note that the light node instanced inside the `GeneratedShadowMap.light` or `ProjectedTextureCoordinate.projector` fields isn't considered a normal light, that is it doesn't shine anywhere. It should be defined elsewhere in the scene to actually act like a normal light. Moreover, it should not be instanced many times (outside of `GeneratedShadowMap.light` and `ProjectedTextureCoordinate.projector`), as then it's unspecified from which view we will generate the shadow map.

Fields `scale` and `bias` are used to offset the scene rendered to the shadow map. This avoids the precision problems inherent in the shadow maps comparison. In short, increase them if you see a strange noise appearing on the shadow casters (but don't increase them too much, or the shadows will move back). You may increase the `bias` a little more carelessly (it is multiplied by a constant implementation-dependent offset, that is usually something very small). Increasing the `scale` has to be done a little more carefully (it's effect depends on the polygon slope).

Figure 4 shows the effects of various `scale` and `bias` values.

For an OpenGL implementation that offsets the geometry rendered

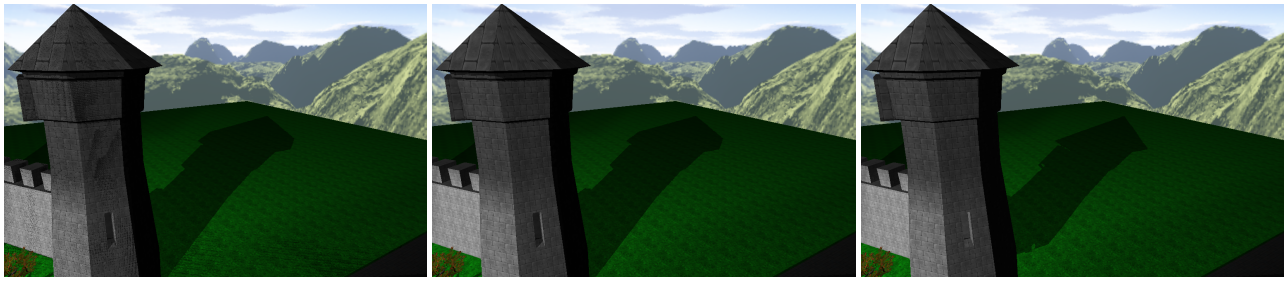

Figure 4: *Various bias/scale values. On the left, they are too small and the shadow caster is erroneously considered to cast shadow on itself (notice the noise on the tower). On the right, they are too large and the shadow is pushed back and shrunk (notice how it doesn't touch precisely the bottom of the tower). In the middle, they are just right (use default values).*

into the shadow map, `scale` and `bias` are an obvious parameters (in this order) for the `glPolygonOffset` call. Other implementations are free to ignore these parameters, or derive from them values for their offset methods.

Field `compareMode` allows to additionally do depth comparison on the texture. For texture coordinate (s, t, r, q), compare mode allows to compare r/q with $texture(s/q, t/q)$. Typically combined with the projective texture mapping, this is the moment when we actually decide which screen pixel is in the shadow and which is not. Default value `COMPARE_R_LEQUAL` is the most useful value for standard shadow mapping, it generates 1 (true) when $r/q <= texture(s/q, t/q)$, and 0 (false) otherwise. Recall from the section 2 that, theoretically, assuming infinite shadow map resolution and such, r/q should never be smaller than the texture value.

When the `compareMode` is set to `NONE`, the comparison is not done, and depth texture values are returned directly. This is very useful to visualize shadow maps, for debug and demonstration purposes — you can view the texture as a normal grayscale (luminance) texture. In particular, problems with tweaking the `projectionNear` and `projectionFar` values become easily solvable when you can actually see how the texture contents look.

For OpenGL implementations, the most natural format for a shadow map texture is the `GL_DEPTH_COMPONENT` (see `ARB_depth_texture`). This makes it ideal for typical shadow map operations. For GLSL shader, this is best used with `sampler2DShadow` (for spot and directional lights) and `samplerCubeShadow` (for point lights). Unless the `compareMode` is `NONE`, in which case you should treat them like a normal grayscale textures and use the `sampler2D` or the `samplerCube` types.

4.5 Projective texture mapping

We propose a new `ProjectedTextureCoordinate` node:

ProjectedTextureCoordinate : X3DTextureCoordinateNode

```
SFNode     [in,out]  projector  NULL
  # [SpotLight, DirectionalLight,
  # X3DViewpointNode]
```

This node generates texture coordinates, much like the standard `TextureCoordinateGenerator` node[1]. More precisely, a texture coordinate (s, t, r, q) will be generated for a fragment that corresponds to the shadow map pixel on the position $(s/q, t/q)$, with r/q being the depth (distance from the light source or the viewpoint, expressed in the same way as depth buffer values are stored in the shadow map). In other words, the generated texture coordinates will contain the actual 3D geometry positions, but expressed in the projector's frustum coordinate system. This cooperates closely with the `GeneratedShadowMap.compareMode = COMPARE_R_LEQUAL` behavior, see the previous subsection.

This can be used in all situations when the light or the viewpoint act like a projector for a 2D texture. For shadow maps, `projector` should be a light source.

When a perspective `Viewpoint` is used as the `projector`, we need an additional rule. That's because the viewpoint doesn't explicitly determine the horizontal and vertical angles of view, so it doesn't precisely define a projection. We resolve it as follows: when the viewpoint *that is not currently bound* is used as a projector, we use `Viewpoint.fieldOfView` for both the horizontal and vertical view angles. When the *currently bound* viewpoint is used, we follow the standard `Viewpoint` specification for calculating view angles based on the `Viewpoint.fieldOfView` and the window sizes. We feel that this is the most useful behavior for scene authors.

When the geometry uses a user-specified vertex shader, the implementation should calculate correct texture coordinates on the CPU. This way shader authors still benefit from the projective texturing extension. If the shader author wants to implement projective texturing inside the shader, he is of course free to do so, there's no point in using `ProjectedTextureCoordinate` at all then.

Note that this is not suitable for point lights. Point lights do not have a direction, and their shadow maps can no longer be single 2D textures. Instead, they must use six 2D maps. For point lights, it's expected that the shader code will have to do the appropriate texture coordinate calculation: a direction to the point light (to sample the shadow map cube) and a distance to it (to compare with the depth read from the texture).

4.6 Define shadow casters

By default, every `Shape` in the scene casts a shadow. This is the most common setup for shadow maps. However it's sometimes useful to explicitly disable shadow casting (blocking of the light) for some tricky shapes. For example, this is usually desired for shapes that visualize the light source position. For this purpose we extend the `Appearance` node:

[1]The reasoning for inventing a new node, instead of extending the existing `TextureCoordinateGenerator`, is that the `projector` field would not be useful for other `TextureCoordinateGenerator` modes.

Additional fields for the Appearance node			
SFBool	[in,out]	**shadowCaster**	TRUE

Note that if you disable shadow casting on your shadow receivers (that is, you make all the objects only casting or only receiving the shadows, but not both) then you avoid some offset problems. The `bias` and `scale` parameters of the `GeneratedShadowMap` become less crucial then.

This field may also be used by other shadow approaches implemented in the X3D browsers. For example, shadow volumes or ray-tracers could use it too.

4.7 How the receiveShadows field maps to the lower-level extensions

Placing a light on the `receiveShadows` list is equivalent to adding the appropriate `GeneratedShadowMap` to the shape's textures, and adding the appropriate `ProjectedTextureCoordinate` to the geometry `texCoord` field. Also, `receiveShadows` makes the right shading (for example by shaders) automatically used.

In fact, the `receiveShadows` feature may be implemented by a simple transformation of the X3D node graph. Since the `receiveShadows` and `defaultShadowMap` fields are not exposed (they do not have accompanying input and output events) it's enough to perform such transformation once after loading the scene. Note that the texture nodes of the shadow receivers may have to be internally changed to multi-texture nodes during this operation.

An author may also *optionally* specify a `GeneratedShadowMap` node inside the light's `defaultShadowMap` field. See the section 4.3 for `defaultShadowMap` declaration. Leaving the `defaultShadowMap` as `NULL` means that an implicit shadow map with default browser settings should be generated for this light. This must behave like `update` was set to `ALWAYS`.

In effect, to enable the shadows the author must merely specify which shapes receive the shadows (and from which lights) by the `Appearance.receiveShadows` field. This way the author doesn't have to deal with lower-level tasks:

1. Using `GeneratedShadowMap` nodes.
2. Using `ProjectedTextureCoordinate` nodes.
3. Writing own shaders.

5 Shading with the shaders

When the shape receiving shadows doesn't have any shaders assigned in the X3D file, we do not require much from the implementation. We only insist that shadows must be visualized in *some* way, from at least one shadow map. In the simplest case, the computed pixel color may be simply set to the pure black when it falls in the shadow. This way even implementations for old GPUs, that have only fixed-function pipeline available, may satisfy our extensions specification.

The encouraged method to make the shadows look nice is to use the shaders. This way shadows have high-quality look, and the shading doesn't require any extra rendering passes. The X3D author has full power of customizing the shadows look when using lower-level `GeneratedShadowMap` approach. When the `receiveShadows` field is used, browsers are strongly encouraged to use internal shaders for nice shading.

Below we present simple examples how to use GLSL (OpenGL Shading Language) shaders to make the shadows look nice. Note that we use GLSL language just as an example. Our extensions are not OpenGL specific, and any shading language is usable with our shadow maps. We extend the X3D specification of the shaders component to pass `GeneratedShadowMap` to shader's `sampler2DShadow` type (for spot and directional lights) and `samplerCubeShadow` type (for point lights). Unless they have `compareMode` set to `NONE`, in which case they map (appropriately) to the `sampler2D` or the `samplerCube`. This applies to all shader languages mentioned in the X3D specification: *OpenGL shading language (GLSL) binding*, *Microsoft high level shading language (HLSL) binding* and *nVidia Cg shading language binding*.

5.1 Basics

In the simplest case, we sample the 2D depth texture using the `shadow2DProj(shadowMap, gl_TexCoord[0])` call. A complete GLSL fragment shader looks like this:

```
uniform sampler2DShadow shadowMap;
void main(void)
{
  float shadow =
   shadow2DProj(shadowMap, gl_TexCoord[0]).r;
  gl_FragColor = gl_Color * shadow;
  /* add some ambient term */
  gl_FragColor += vec4(0.25, 0.25, 0.25, 1);
}
```

Figure 5: *Close up shadows on the tree. Notice that leaves (modeled by alpha-test texture) also cast correct shadows.*

Figure 6: *The previous tree once again, this time with percentage closer filtering (16 samples). Note the soft look of the shadows.*

This is a very simple shader, and a very crude one. Next we will describe how to improve it.

5.2 Improvements

The shader code in the previous section scales the `gl_Color` value by the shadow amount of a single light. This isn't a correct solution, as `gl_Color` (calculated by the vertex shader or the fixed-function pipeline) contains the contribution from *all scene lights*.

A better solution would be to calculate the whole lighting inside the fragment shader [2]. Then the `shadow` value may be used to scale only the contribution of the appropriate light. Following this idea, we could also use several shadow maps, each one from a different light source. Shader could then calculate lighting with correctly combined shadows from all the light sources.

Another problem is that the shadow map may be sampled with positions outside of the $(0,0) - (1,1)$ square. This isn't a problem for spot lights since they do not shine outside of their cone, so the value of `shadow` there doesn't matter — it will be multiplied by black color anyway. But for directional lights it remains important. Outside of the $(0,0) - (1,1)$ square, the clamping of the texture coordinates will stretch the shadows over unrelated scene parts. We would like to consider everything that is outside of the shadow map as always in the shadow. This may be done by inserting (before the `shadow` value is used in the multiplication) the following check into the previous shader code:

```
vec2 shadowMapCoord =
  gl_TexCoord[1].st / gl_TexCoord[1].q;
if (shadowMapCoord.s < 0.0 ||
    shadowMapCoord.s > 1.0 ||
    shadowMapCoord.t < 0.0 ||
    shadowMapCoord.t > 1.0)
  shadow = 0.0;
```

5.3 Percentage closer filtering

When the light is far from the shadow receiver, and the user's avatar looks at the shadow closely, then a single shadow map pixel corresponds to many pixels on the screen. This means that the shape of the shadow map pixels is unfortunately visible on the screen. *Percentage closer filtering* [Bunnell and Pellacini 2004] hides these artifacts by averaging the *results* of the shadow tests. A trivial implementation inside GLSL shader may be seen here: `https://vrmlengine.svn.sourceforge.net/svnroot/vrmlengine/trunk/demo_models/shadow_maps/shadow_map_pcf4.fs`.

6 Implementation notes

An open-source implementation of these extensions is available in our engine on `http://vrmlengine.sourceforge.net/`. You can test it for example with our `view3dscene` tool. An example VRML/X3D files with the shadow maps and the GLSL shaders are available in our test suite on `http://vrmlengine.sourceforge.net/demo_models.php` (inside `shadow_maps/` subdirectory).

Our implementation uses the basic OpenGL tools: we render to the depth texture using the framebuffer object, and we set up projective texturing by OpenGL `glTexGen` procedures. We perform the comparison to determine if the shadows are black by the `ARB_shadow` OpenGL extension. Everything needed is available in pretty much every sensible OpenGL implementation (even in the pure software version of Mesa3D).

The case when `update` is set to `ALWAYS` is optimized. The shadow maps are regenerated only if the geometry of the scene changes. This means that mere camera movements, or animation of the materials or textures do not cause an unnecessary update overhead. An other potential optimization would be to update shadow maps only once for a couple of frames, or once for a fraction of a second.

We do not handle shadow maps from `PointLight` sources yet.

The `receiveShadows` field is implemented as a transformation of the X3D nodes graph. You can try it by the `view3dscene` menu option *Handle receiveShadows by shadow maps*. Currently, the effect of this transformation may be visible to the author in some uncommon situations, like when using a script traversing the nodes graph.

We have also implemented the shadow volumes algorithm [Kamburelis 2006] and a software ray-tracer. They both honor the `shadowCaster` field. Our ray-tracer treats everything as a shadow receiver. Our current shadow volumes implementation is slightly limited, and requires special lights setup, also treating everything as a shadow receiver. Non-manifold shadow casters are detected but currently still used by the shadow volumes implementation — this is useful for special non-manifold shapes in some scenes, at the risk of showing ugly artifacts from particular camera angles.

7 Conclusion

We have presented a number of extensions to the existing X3D specification. Together, they allow the authors to control the shadow maps behavior on the 3D scenes. Authors can also use projective texturing as an independent feature, for example to cast normal colored textures from the light sources or viewpoints. Generated shadow map textures can be also visualized as a grayscale 2D textures, which is very useful for debugging shadow map depth problems.

Most important (and scene-dependent) parameters of the shadow map generation can be set in the X3D code, while the implementation is delegated to hide most of the dirty work. Natural implementation in both fixed-function and shader pipelines is possible. Shadow maps defined this way cooperate nicely with user's custom shader code.

8 Future work

Shadow mapping, as well as many other effects, would benefit from X3D browser being able to use it's own internal shader code. For example, advanced bump mapping variants (like *steep parallax bump mapping with self-shadowing* [McGuire and McGuire 2005]) can be sensibly implemented only using the shaders. Current implementation of the bump mapping extension in our engine forces the author to make a difficult choice: *either* easily „turn on the bump mapping" (our bump mapping extensions require only to provide the height and normal maps [Kamburelis 2008]) and resign from custom shaders *or* resign from using our comfortable extensions and implement bump mapping yourself with own shaders. The simple `receiveShadows` extension described in the section 4.1 has the same disadvantage: browser's internal shaders have to override author's custom shaders in this case.

[2] Or pass contributions from the separate lights as separate variables from the vertex shader.

It's worth exploring how to merge such systems. Ideally, author should be able to write his own shader code, and at the same time the implementation must be able to implement some effects through internal shaders. Author must have the ability to selectively use or override all the effects of the browser.

A flexible approach to cooperate between user and browser shader code (preferably an approach mapping to the various shader languages: GLSL, Cg, HLSL, also CgFX) seems like a mighty extension for X3D.

Acknowledgements

The author would like to thank the many people that encouraged the development and helped to test our VRML/X3D engine. In particular Victor Amat was experimenting with *Screen Space Ambient Occlusion* and uncovered many quirks in the implementation.

Andrzej Łukaszewski did the proofreading of this paper and suggested a lot of the improvements.

The 3D scenery with trees and towers visible on many screenshots in this paper was constructed with the help of the assets from `http://opengameart.org`, a great site with good quality 3D assets on clear open-source licenses. See the `https://vrmlengine.svn.sourceforge.net/svnroot/vrmlengine/trunk/demo_models/shadow_maps/sunny_street/` for the full scene code in X3D, helper transformation program and the `README.txt` file with detailed credits.

References

BITMANAGEMENT, 6.2. BS Contact VRML/X3D 6.2. `http://www.bitmanagement.de/support/developer-area/whats-new/bs-contact-vrml/x3d-62`, see the *Projective Texture Mapping* section.

BITMANAGEMENT, 7.2. BS Contact 7.2 - Release Notes - Advanced realtime shadows. `http://www.bitmanagement.de/developer/contact/relnotes72.html#shadow`.

BUNNELL, M., AND PELLACINI, F. 2004. *GPU Gems: Shadow Map Antialiasing*. ch. 11. `http://developer.nvidia.com/object/gpu_gems_home.html`.

EVERITT, C., REGE, A., AND CEBENOYAN, C., 2001. Hardware Shadow Mapping. `http://developer.nvidia.com/object/hwshadowmap_paper.html`.

EVERITT, C., 2001. Projective Texture Mapping. `http://developer.nvidia.com/object/Projective_Texture_Mapping.html`.

EVERITT, C., 2001. Shadow Mapping. `http://developer.nvidia.com/object/shadow_mapping.html`.

KAMBURELIS, M., 2006. Shadow volumes extensions. `http://vrmlengine.sourceforge.net/kambi_vrml_extensions.php#section_ext_shadows`.

KAMBURELIS, M., 2008. Bump mapping extensions. `http://vrmlengine.sourceforge.net/kambi_vrml_extensions.php#section_ext_bump_mapping`.

MCGUIRE, M., AND MCGUIRE, M. 2005. Steep parallax mapping. *I3D 2005 Poster*. `http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html`.

OCTAGA. Octaga Nodes. `http://www.octaga.com/Developer/OctagaNodes.pdf`, pages 54-55.

WEB 3D CONSORTIUM, 2008. Extensible 3D (X3D) Graphics Standard. ISO/IEC 19775-1.2:2008; see `http://web3d.org/x3d/specifications/`.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH '78)*, 270–274.

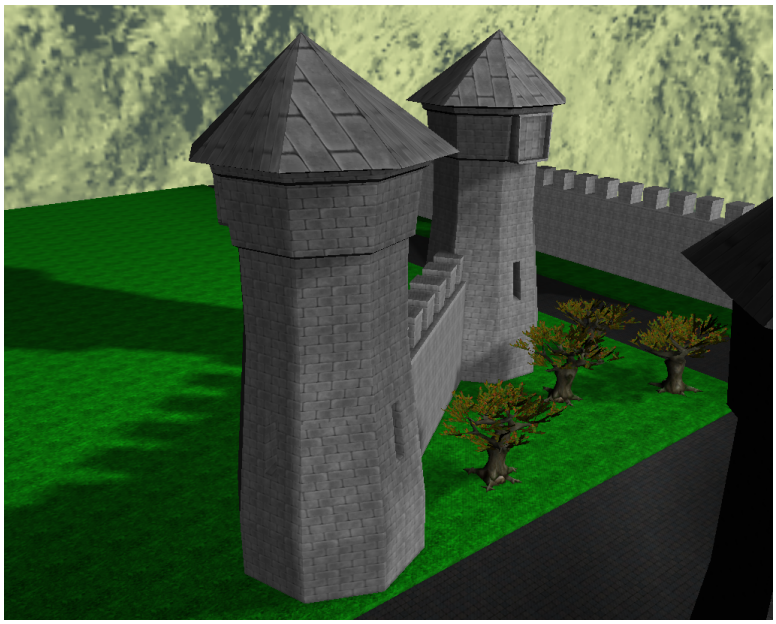

Figure 7: *Yet another screenshot with nice shadow maps.*