# Real-Time Shadows On Complex Objects

August, 2000

**Gabor Nagy**
Sony Compter Entertainment America
Gabor_Nagy@Playstation.sony.com

This gem presents an efficient algorithm capable of creating realistic shadows in real-time applications.
The algorithm can take advantage of today's fast texture-mapping and 3D transformation hardware.

## Introduction

Shadows are among the most important depth-cues in human vision. In computer graphics, they can give the 'final touch of realism' to an image.
Without cast shadows, even with realistic lighting and texturing effects, computer generated images look artificial, the objects appearing to "float in space" even when they are laying on a surface.
This lack of sense in relative position and depth is especially apparent when the camera is not moving (no parallax information).

Until recently, only computationally expensive algorithms, such as ray-tracing and radiosity could produce accurate shadows, where both the objects casting the shadows and the ones receiving them are of arbitrary complexity.

The algorithm presented here is optimized for real-time applications. It provides a very good balance between realism and rendering performance while being easily extendable to all situations.

With the always performance-hungry game programmer in mind, the gem will highlight points when significant optimization for performance is possible, using a hardware feature.

While some of the basic ideas in this gem have been around for a while, most papers describing them don't deal with some of the important implementation details.

## Light source, Blocker object, Receiver object

Consider the simple example shown on Figure 1.

The torus ("blocker object" or "blocker") blocks some of the light coming from the light source, casting a shadow on the wall.

The wall receives the shadow, or "lack of light", therefore it is called the "receiver object", or "receiver".

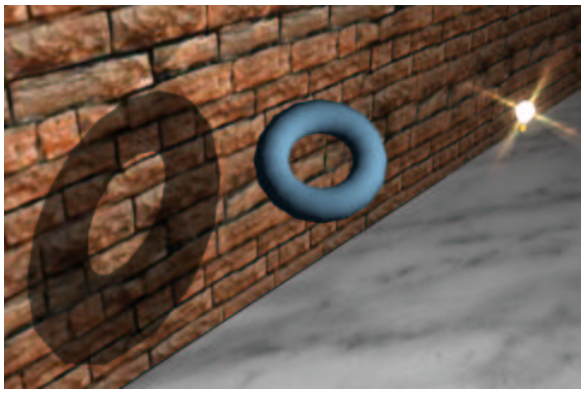

Figure 1: Shadow, Receiver, Blocker and Light source

If the light source is a point light (infinitely small), the blocker object will block the light of that light source in a well-defined volume, usually referred to as a "*shadow-volume*" (see Figure 2).

A shadow will be created on a receiver object where its surface intersects with the *shadow-volume*.

As Figure 2. shows, the *shadow-volume* has a cone or trapezoid-like shape, starting at the blocker object and continuing to infinity.
While the *shadow-volume* really starts at the contours of the blocker object, its cone-like shape originates from the light source.

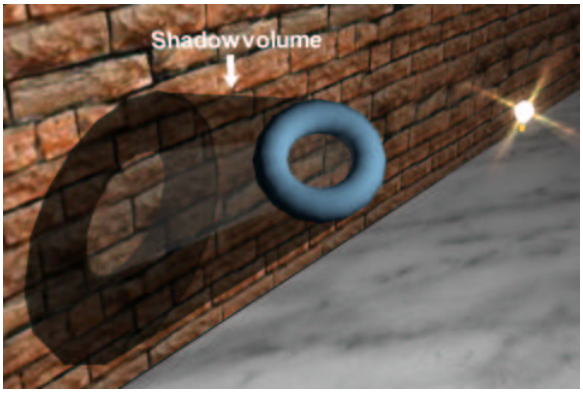


Figure 2: the "shadow volume"

Let's examine how the cross section of the shadow volume changes as we get further from the light source.
We'll call the point on the blocker's surface that is nearest to the light source "$P_{nl}$" and the one farthest from it: $P_{fl}$.

We can divide the shadow volume into 3 sections:
1) Between the light source and $P_{nl}$
2) Between $P_{nl}$ and $P_{fl}$
3) From $P_{fl}$ to infinity

It's easy to see that in sections 1) and 3), the cross-section of the *shadow-volume* will have a constant shape, but it will increases in size as we get further from the light source.

Because of the above, unless one or more receiver objects are in section 2), the *shadow-volume* can be accurately modeled by projecting a 2-dimensional mask form a point (the position of the light source).
Consequently, using the same projection, we can "map" this 2D mask on the receiver objects to define the shadowed areas!

This 2D mask is called the "*shadow-map*", and it can be simply derived by drawing the blocker object's silhouette as seen from the light source.

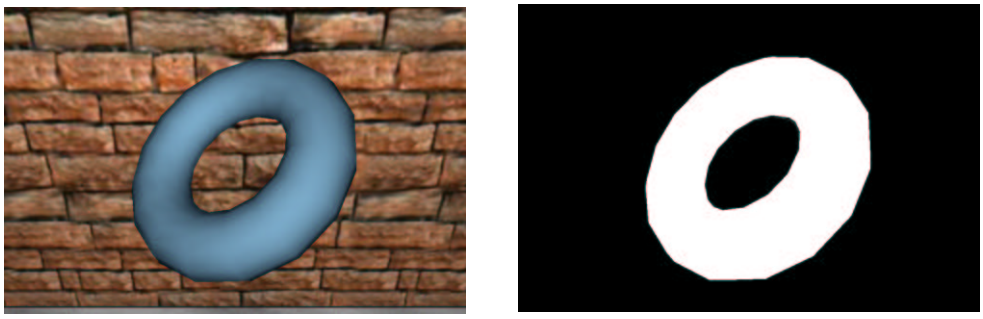

Figure 3a and 3b: The blocker object as seen from the light source (left) and its silhouette (right)

Notice that we can not see the shadow cast by the torus on Figure 3a., because the torus exactly obscures it!
This is in fact a good indication that indeed, we can just use a properly projected 2D image or mask (see Figure 3b.) to define the shadow volume.

This method is usually referred to as "projective shadow-mapping".

## The objectives of this gem

To draw shadows using the method introduced above, we need to do the following:

1. Create a shadow-map for each light/blocker object pair
2. Calculate the *shadow-map* (texture) coordinates to use on the receiver object's vertices
3. Render the receiver objects with the *shadow-map* applied as a 2D texture

## 1. Creating the shadow-map

We will set up a perspective projection originating at the light source.
This will project the blocker object onto a virtual *screen-plane* between the light source and the blocker object, yielding the *shadow-map* as seen on Figure 4.

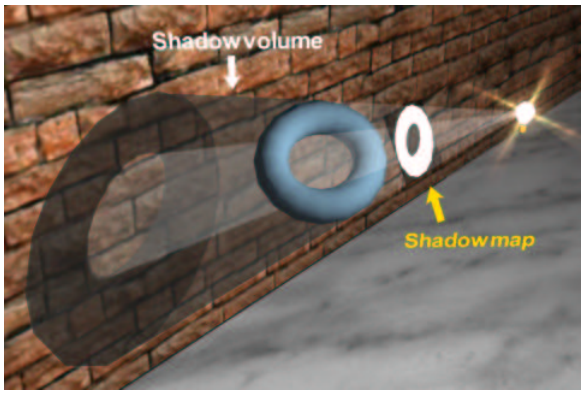


Figure 4: the *shadow-map* projection

### 1.1. The *light coordinate-system*

First, we will define a new coordinate system with its origin at the light source and its Z axis pointing at the blocker object.
The Z axis of this coordinate-system will determine the center line of the perspective projection, while its XY-plane will define the the orientation of the screen-plane we will project the *shadow-map* on.
If we transform the blocker object into this *light* coordinate-system, we can easily project it onto this plane.

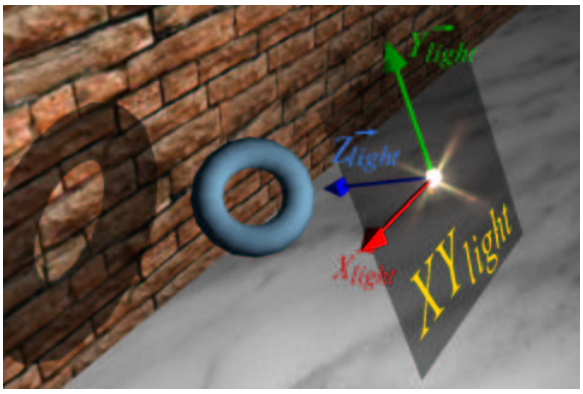


Figure 5: the *light* coordinate-system

To define an arbitrary, non-scaling coordinate-system, we need to know the position of its origin and its orientation.

We already know the position of the origin: it's the position of our light source.

We will describe the orientation of the light coordinate system with the direction of its 3 axes:
$X_{light}$, $Y_{light}$ and $Z_{light}$ , all 3D unit-vectors in *world* coordinates.

### 1.1.1. Finding $Z_{light}$

Starting with the $Z_{light}$ axis, we can easily find $X_{light}$ and $Y_{light}$.

$Z_{light}$ will be a direction vector that starts from the light source and points at the blocker object.

Let's assume that the blocker object is polygonal, and we have an array of all the polygon vertices that are used in rendering this object.
Now we have a set of "target" points in 3D space (the vertices of the blocker) and another point: the position of the light source.

A fast and efficient way of getting "a good" direction vector is to average the vectors starting from the light source and pointing to each vertex.

We will call it the Mean Direction Vector, or *MDV*:

$$\vec{MDV} = \frac{\sum_{i=1}^{N_v} (V_i - P_{light})}{N_v}$$

Where $N_v$ is the number of vertices considered in the blocker and $P_{light}$ is the position of the light source.

Normalizing *MDV* (making its length 1.0) will yield $Z_{light}$:

$$\vec{Z_{light}} = \frac{\vec{MDV}}{\left|\vec{MDV}\right|}$$

**Optimization tip #1**
Since we will normalize *MDV* anyway, we don't have to divide the sum of light-to-vertex vectors by *Nv*, saving one divide operation.
We can also calculate $P_{light}$*$Nv$ in advance and avoid the "- $P_{light}$" in the vertex loop, because:

$$\frac{\sum_{i=1}^{N_v} (V_i - P_{light})}{N_v} = -P_{light} N_v \frac{\sum_{i=1}^{N_v} V_i}{N_v}$$

Here is the C code to get $Z_{light}$:

```
typedef struct
{
 E3dType     X,Y,Z;
 short Flags;
} E3dVertex;


void ShadowMatrix(Matrix LBlockerLocalToWorldMatrix)
{
 unsigned long      LVn, LVC, LN, LC;
 float       Mx, My, Mz, LPlightX, LPlightY, LPlightZ,
 float       LMDVX, LMDVY, LMDVZ,             // Mean Direction Vector
 float       LZlightX, LZlightY, LZlightZ,    // Zlight vector X, Y and Z

// Initialize Mean Direction Vector to (0.0, 0.0, 0.0)
//
 LMDVX = LMDVY = LMDVZ = 0.0;

 Lvertex = LMesh->Vertices;

// Average vertex-to-light vectors
//
 LVn = LMesh->NumOfVertices;

 LMDVX = LPlightX * LVn;
 LMDVY = LPlightY * LVn;
 LMDVZ = LPlightZ * LVn;

 for(LVC = 0;L VC < LVn; LVC++, LVertex++)
 {
  Mx=LVertex->X; My=LVertex->Y; Mz=LVertex->Z;
  E3dM_MatrixTransform3x4(LBlockerLocalToWorldMatrix, LX, LY, LZ);

  LMDVX -= LX;
  LMDVY -= LY;
  LMDVZ -= LZ;
 }

// Normalize Mean Direction Vector (MDV)
//
 LVF = sqrt(LMDVX*LMDVX+LMDVY*LMDVY+LMDVZ*LMDVZ);
 LVF = 1.0 / LVF;          // We can save 2 divisions by doing this in advance...

 LZlightX = LMDVX * LVF;
 LZlightY = LMDVY * LVF;
 LZlightZ = LMDVZ * LVF;

.
.
}
```

`E3dM_MatrixTransform3x4` is a macro function that transforms a 3D vector given by *Mx, My* and *Mz* with a 3x4 matrix (actually the top-left part of a 4x4 matrix).

For the rest of the source code, please refer to the example program on the companion CD-ROM.

### 1.1.2. Finding $X_{light}$ and $Y_{light}$

The projection to map the *shadow-map* texture on the receiver object will be the same as the one used to draw the *shadow-map,* therefore the orientation of the *shadow-map* (rotation

around $Z_{light}$) does not matter. In other words: rotating the *XY-light* plane around $Z_{light}$ will not make any difference.
This means that for the $X_{light}$ axis, we can use any unit vector that is perpendicular to $Z_{light}$ (see Figure 5.).

We can get a vector like that as the cross-product of $Z_{light}$ and any other vector that is not parallel with $Z_{light}$.
Let's call this other "helper" vector $V$.

We know that at least two of the X, Y or Z axes of the *world* coordinate system will meet this criteria, so for simplicity, we will use a unit vector $V$(x,y,z) with one coordinate being 1, the others 0.

A vector's largest component (X, Y or Z) will determine it's "dominant" direction, therefore to get a vector that points "far enough away" from $Z_{light}$,, we will set the component of $V$ to 1 that has the smallest absolute value in $Z_{light}$.
This will eliminate "float"-precision worries when performing a vector cross-product operation on $Z_{light}$ and $V$.
For example:
if $Z_{light}$=(0.381, 0.889, 0.254), $V$ will be: (0.0, 0.0, 1.0) = $Z_{world}$
if $Z_{light}$=(-0.889, 0.254, 0.381), $V$ will be: (0.0, 1.0, 0.0) = $Y_{world}$
and so on.

The cross-product of $Z_{light}$ and $V$ will yield a third vector that is perpendicular to both of them.
After normalization, this will yield $X_{light}$, the X axis of the light coordinate system:

$$\vec{X}_{light} = \frac{\vec{Z}_{light} \times \vec{V}}{\left| \vec{Z}_{light} \times \vec{V} \right|}$$

With $X_{light}$ and $Z_{light}$ given, the $Y_{light}$ axis is just another cross-product away:

$$\vec{Y}_{light} = \vec{Z}_{light} \times \vec{X}_{light}$$

Note that this will give us a unit vector, so we don't have to normalize $Y_{light}$, because:

$$\left| \vec{X}_{light} \right| = 1 \text{ and } \left| \vec{Z}_{light} \right| = 1 \text{ and } \vec{Z}_{light} \perp \vec{X}_{light} \Rightarrow \left| \vec{Z}_{light} \times \vec{X}_{light} \right| = 1$$

With $X_{light}$, $Y_{light}$ and $Z_{light}$ and $P_{light}$ known, we can create the matrix that will transform a point from *world* coordinates to *light* coordinates by simply filling in these values:

$$M_{WorldToLight} = \begin{bmatrix} X\ of\ \vec{X}_{light} & X\ of\ \vec{Y}_{light} & X\ of\ \vec{Z}_{light} & 0.0 \\ Y\ of\ \vec{X}_{light} & Y\ of\ \vec{Y}_{light} & Y\ of\ \vec{Z}_{light} & 0.0 \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

This is why we used $X_{light}$, $Y_{light}$ and $Z_{light}$ to describe the orientation of the *light* coordinate-system.

The next step is to pre-multiply this matrix with the blocker object's *local-to-world* matrix. This will give us the *local-to-light* matrix for the blocker.

$$M_{BlockerLocalToLight} = M_{BlockerLocalToWorld} * M_{WorldToLight}$$

As the name implies, the above matrix will transform a point defined in the *local* coordinate system of the blocker into the *light coordinate-system.*

Such transformed X and Y coordinates will define the *parallel* or *orthogonal* projection of the blocker object onto the *shadow-map* plane (which is parallel with the XY-plane of the *light* coordinate system).

### 1.2.The perspective projection

To make this a perspective projection, we will need a field-of-view, or the X and Y "projection ratios".
We can find the projection ratios ($R_X$ and $R_Y$) for each vertex of the blocker object by transforming the vertex with $M_{BlockerLocalToLight}$ and dividing the resulting X and Y coordinates by the resulting Z coordinate (see Figure 6.).

$$R_Y = \left| \frac{Vtx_Y}{Vtx_Z} \right|$$

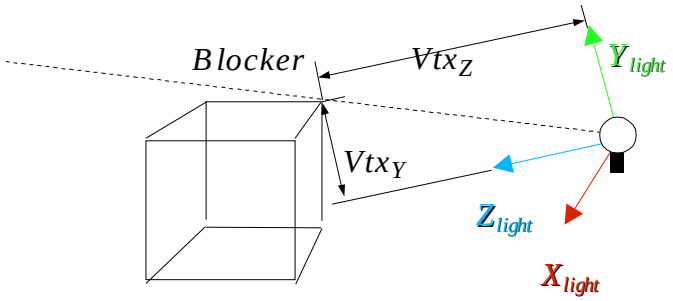


Figure 6.: Projection ratios

### 1.2.1. Adaptive projection

We could use a single ratio for the projection, but that would lead to the blocker object's silhouette changing size in the *shadow-map* if we move the light source closer or farther

away from it.
The same problem would arise if we change the size of the blocker or if the light source "looks at it" from a different angle.
This could result in a tiny image of the blocker in the middle of the *shadow-map*, or an over-sized image that doesn't fit in the *shadow-map* (see Figure 7.).
In the first case (Figure 7a.) we get a low resolution *shadow-map* with bad artifacts on the receiver objects. This would be a wasting of *shadow-map* memory.
The latter (Figure 7b.) will cause incorrect shadow shapes and possibly "shadow-leaking" (see section 2. texture-clamping).
In this case the *shadow-map* memory would not be enough.

Figure 7.: Non-adaptive (7a and 7b) and adaptive blocker projection (7c)

Instead of one fixed value, we will use the largest $R_X$ ($R_{Xmax}$) as the horizontal ratio for the projection, while the largest $R_Y$, ($R_{Ymax}$) will give the vertical ratio.
This makes the perspective projection *adaptive* for both the X and Y direction, meaning that the blocker's silhouette will always properly fill the *shadow-map*, making the best use of all the pixels in it.

This is very important, as we want to use the minimum necessary texture size, because:
- Texture memory is always a scarce resource and the maximum texture size might be limited by other factors.
- On some hardware, after drawing the *shadow-map* image, we have to transfer it from the frame-buffer to a dedicated texture memory and the speed of this transfer will be limited by bus and memory bandwidth.

Now we can fill out a standard perspective projection matrix for the blocker object:

$$M_{BlockerProjection} = \begin{bmatrix} \frac{1}{R_{Xmax}} * SMapWidth & 0 & 0 & 0 \\ 0 & \frac{1}{R_{Ymax}} * SMapHeight & 0 & 0 \\ 0 & 0 & \frac{Z_{far}+Z_{near}}{Z_{near}-Z_{far}} & - \\ 0 & 0 & 2\frac{Z_{far}Z_{near}}{Z_{near}-Z_{far}} & 0 \end{bmatrix}$$

Where: *SMapWidth* and *SMapHeight* are the horizontal and vertical resolution of the *shadow-map* in pixels.
$Z_{near}$ and $Z_{far}$ are the distances of the "Near" and the "Far" clipping planes of the viewing frustum from the light source.

Pre-multiplying this matrix with $M_{BlockerLocalToLight}$ will yield the 4x4 matrix that will perform a perspective projection from *world* coordinates to *shadow-map* space:

$$M_{BlockerLocalToShadowMap} = M_{BlockerLocalToLight} * M_{BlockerProjection}$$

In OpenGL, we can simply load an identity matrix into the PROJECTION matrix and $M_{BlockerLocalToShadowMap}$ into the MODELVIEW matrix and start drawing the *shadow-map* using the blocker's local coordinates.

**Optimization tip #2:**
To decrease the time it takes to create the *shadow-map*, we can use 2 or 3 different versions of the blocker object geometry for the different rendering stages:

- To set up the *shadow-map* projection: blocker geometry with a minimum number of vertices. We won't need connectivity data (e.g.: polygons) or normal-vectors here.
  All that matters is that no polygons of the blocker should be outside the projected shape of this volume, no matter the angle we look at it from as that would draw on the 1-pixel edge of the shadow-map, ruining texture-clamping and causing "shadow leaking".
  We might even use a "good" bounding volume such as the blocker's bounding-box. This could eliminate the need to compute *MDV* (just use a vector from the light source to the center of the bounding box).

- To draw the *shadow-map*: blocker geometry with minimum or no *surface-detail*, but necessary *contour-detail.*

- And of course, to draw he blocker object, we need the geometry with all the *surface-detail* and surface properties (e.g.: normal vectors) to make the object look "spiffy".

**Optimization tip #3:**
If the rendering engine is programmable, we can use a very simple (and possibly fast) renderer code to draw the *shadow-map*:
- No lighting needed, a simple "flat-color" renderer will do.
- No clipping needed (the blocker's image will always fit in the *shadow-map*!).
- No depth-testing (Z-buffering) needed.

## 2. Projecting the *shadow-map* on a receiver object

Now we have a *shadow-map* associated with a blocker object and a light source. This *shadow-map* can be projected on any number of receiver objects, and because it will be applied as a texture, the receiver objects can have any complex shape (curves, holes, ridges etc.).

As mentioned before, we will use the same projection to project the *shadow-map* on a receiver as we used to draw the *shadow-map*.
The only differences are the image offset and scaling factors, because we will use the 0.0...1.0 coordinate range as opposed to the 0...*SMapWidth* or 0...*SMapHeight* ranges.

This is the appropriate projection matrix:

$$M_{ReceiverProjection} = \begin{bmatrix} \frac{0.5}{R_{Xmax}} & 0 & 0 & 0 \\ 0 & \frac{0.5}{R_{Ymax}} & 0 & 0 \\ -0.5 & -0.5 & \frac{Z_{far}+Z_{near}}{Z_{near}-Z_{far}} & -1 \\ 0 & 0 & 2\frac{Z_{far}Z_{near}}{Z_{near}-Z_{far}} & 0 \end{bmatrix}$$

### 2.1. Texture coordinates and *shadow-map* coordinates

The *shadow-map* will be an image with a finite number of pixels and integer coordinate values, for example: 256x256.
However, texture coordinates are usually normalized floating point values, meaning that the range 0.0...1.0 will refer to pixel coordinates 0..255 horizontally and 0..255 vertically.

So what happens outside the 0.0...1.0 range?
We have to make sure that the texture pixel (texel) used on the receiver will be the color used for "no shadow" (black on Figure 7.).

On most 3D hardware with texture-mapping, you have at least 2 options:
- Texture repeat: outside the 0.0...1.0 range, the texture is simply repeated, so for example

in the -1.0...0.0 range of texture coordinates will produce the same image as the 0.0...1.0 range.
- Texture clamping: the pixel on the edge of the texture image is repeated everywhere outside the 0.0...1.0 range, or you can define a specific "border color" that will be repeated outside the normal range.

It's easy to see that we will have to use texture clamping, because we want a uniform effect on the receiver object outside the 0.0...1.0 texture coordinate range.

Texture-clamping will effectively save us the testing of the receiver objects for intersection with the *shadow-volume*.

Because not all 3D hardware and API provide a separate texture border color, we have to leave a 1 pixel thick border on the *shadow-map*.
To make sure that nothing is drawn in this border when rendering the blocker object, we have to slightly decrease the projection ratios.

## 3. Rendering the receiver objects

There are many different ways to draw the object receiving the shadow. The two most common methods are:

- Single-pass rendering:
  If there is no other texture on the receiver object, we can draw it in one pass, applying a black on white *shadow-map* as a texture and using the light source to illuminate the object.

- Multi-pass rendering with subtractive blending:
  If a receiver already has a texture on it and the hardware doesn't support multi-texturing, we will need multiple passes:
  - Draw receiver normally.
  - Draw shadow-pass with subtractive pixel-blending, using a white on black *shadow-map*.
    This will successively decrease the surface color intensity where there is a shadow cast.
    Use "GREATER-OR-EQUAL" or "LESS-THAN-OR-EQUAL" Z comparison functions for drawing multiple passes. This way if you pass the same primitive, it will overwrite or blend the current pass with the previous one.

For a description of pixel-blending, please refer to the "Convincing Glass For Games" Gem in this book.

## 4. Extensions and enhancements to the basic algorithm

Simplicity and high performance usually comes at a price.
The presented projective shadow-mapping algorithm is no exception from that rule: it has some limitations.
However, most of these limitations are very easy to overcome and the algorithm can be extended to handle most cases.

### 4.1. Back-face shadow elimination

One side-effect of projective shadow-mapping is that it will normally map a shadow on the side of the receiver facing away from the light source.

We can correct this by either:
1. determining if a triangle is facing away from the light source and if it is, we can assign out-of-range *shadow-map* coordinates for all of its vertices (the example code on the CD does this).

2. setting up the rendering of the receiver in such a way that it (the receiver) is completely black on the side facing away from the light source (no ambient lighting)
   This is the proper method, because it is closer to what happens in reality.
   However if there is more than 1 light source in the scene, the "back" face of the blocker can be lit by another one.
   In this case, we will have to use multi-pass rendering and add the ambient light and light coming from other light sources in separate drawing passes.

### 4.2. Receiver is behind light source (light is between blocker and receiver)

You have to explicitly check for this case and not map a shadow on the receiver object.

### 4.3. Mutliple light sources, one blocker, one receiver

This case needs the use of multi-pass rendering with subtractive blending on the receiver object.
Use a receiver rendering pass for each *shadow-map*.
The multiple passes will successively decrease the intensity (RGB values) in the shadowed areas on the surface of the receiver, making even the shadow intersections look correct.

### 4.4. One light source, mutliple blockers, one receiver

This case also needs multiple passes. There is one difference though: the cumulative effect of shadow intersections is incorrect, because the two blockers block the light of the *same* light source.
Use the stencil-buffer to not draw in the screen area where there is already a shadow drawn.

References

James Blinn. Me and My (Fake) Shadow. Jim Blinns Corner, pages 53-61, January 1988.

Foley, et al. Computer Graphics Principles and Practice pages 745-753, Addison Wesley, Second Edition 1990.

David Blythe, Tom McReynolds. Programming with OpenGL: Advanced Rendering. SIGGRAPH '96 Course Notes. August 1996.

Paul Heckbert, Michael Herf. Fast Soft Shadows. SIGGRAPH '96 Visual Proceedings, page 145. Aug. 1996.

Paul Heckbert, Michael Herf. Simulating Soft Shadows with Graphics Hardware. CMU-CS-97-104, CS Dept, Carnegie Mellon U., Jan. 1997.