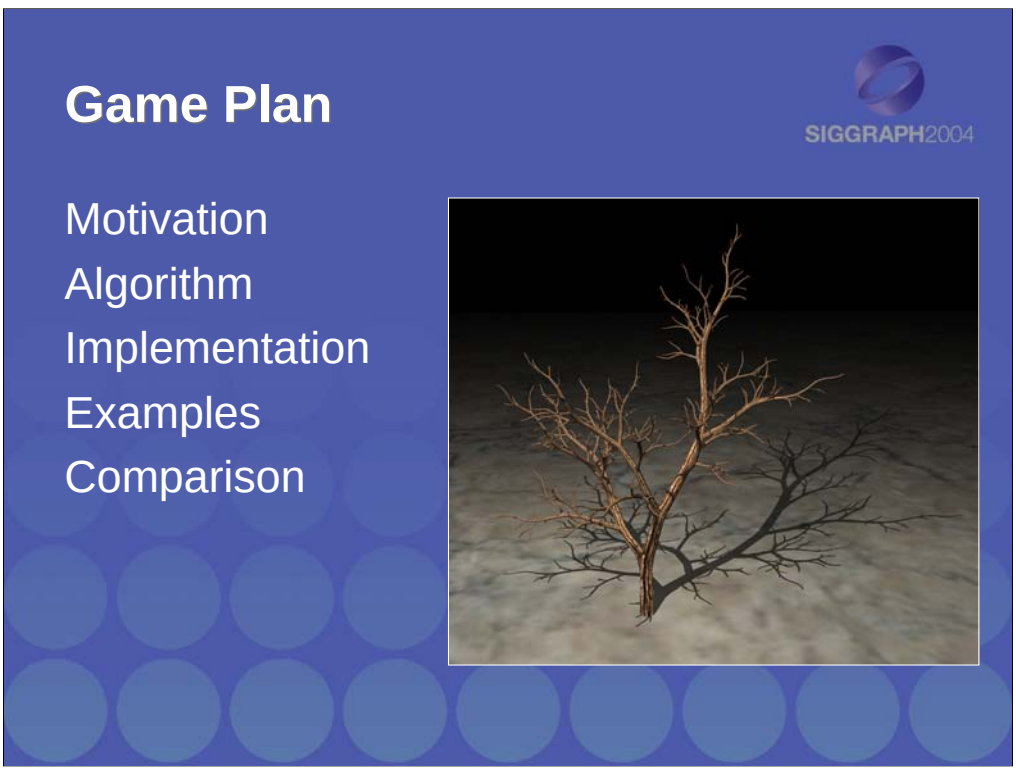

SIGGRAPH2004
Shadow Silhouette Maps
Eric Chan
Massachusetts Institute of Technology
CSAIL

Game Plan

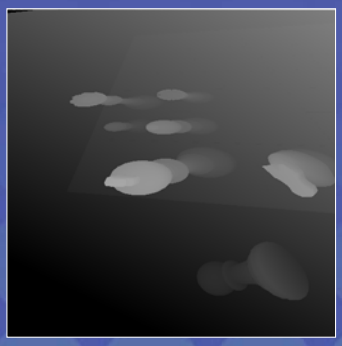


Motivation
Algorithm
Implementation
Examples
Comparison

Motivation

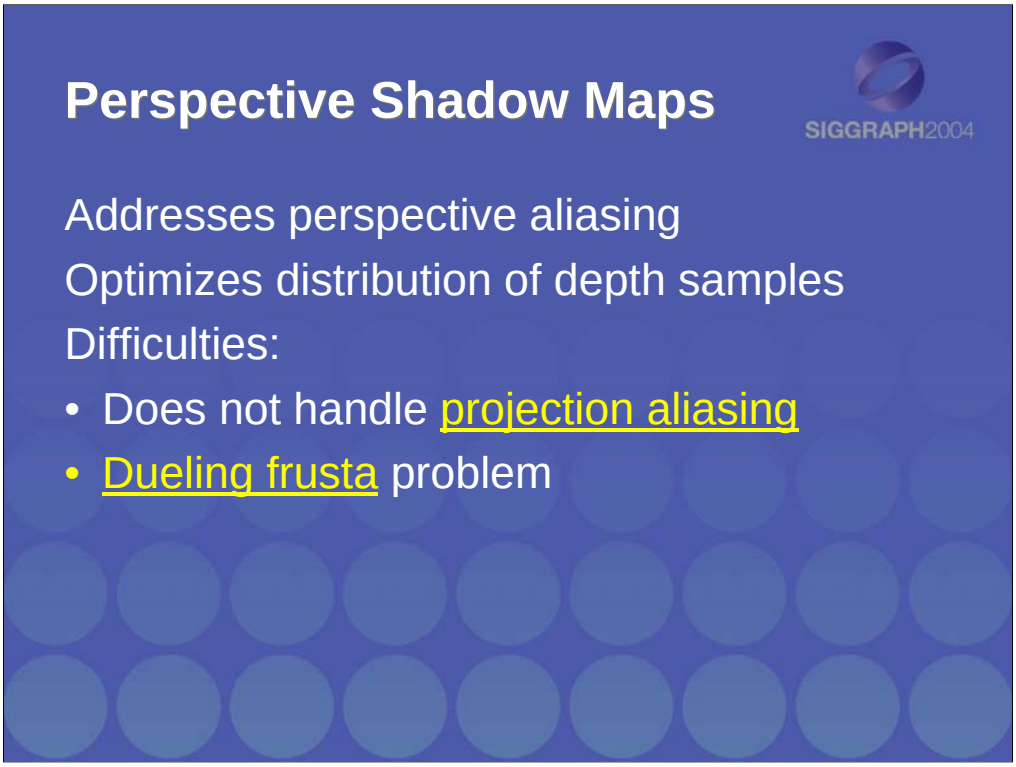
Why another shadow algorithm?

Why not use perspective shadow maps?

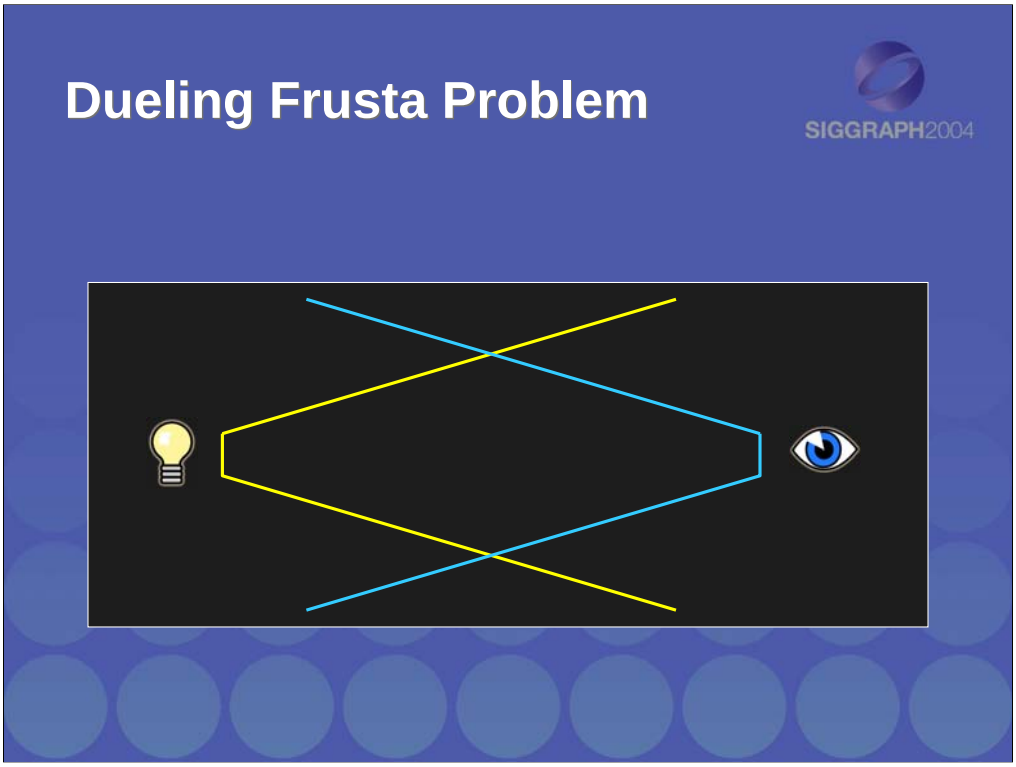


Stamminger and Drettakis, SIGGRAPH 2002

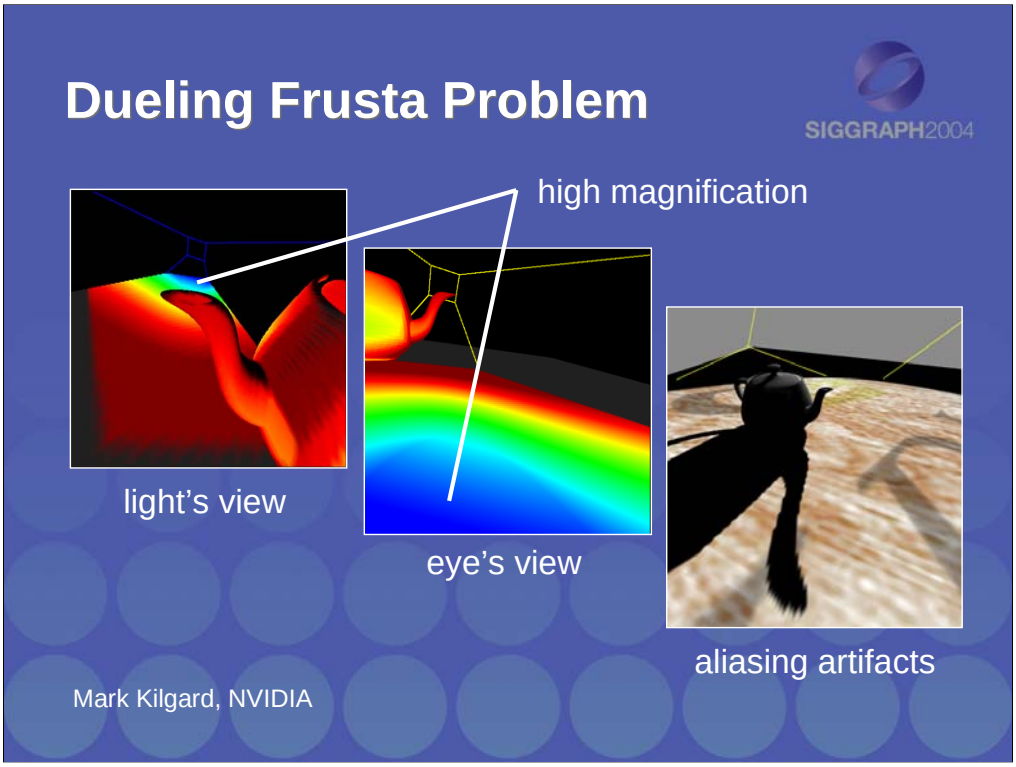
In a nutshell, the topic of this session – the shadow silhouette map – is an extension to the shadow map algorithm that addresses aliasing problems. Perspective shadow maps, described in another session, address the same problem. So why are we bothering with this algorithm if we already have that one? It turns out that the two approaches have different tradeoffs. Hopefully these tradeoffs will become clear as we proceed.



There are two types of aliasing caused by shadow maps: perspective and projection aliasing. See Marc Stamminger's session on Perspective Shadow Maps for a discussion of these aliasing types. Perspective Shadow Maps (PSM) only addresses perspective aliasing, and then only in some cases. In particular, the PSM optimizes the distribution of the depth samples so that more samples are located toward the viewer. However, when the light and camera roughly face each other, PSMs don't work as well. This scene configuration is called the "dueling frusta" problem.



Here is a diagram that shows the light and camera frusta facing each other.



Here is a visualization of what happens from both the light's view and the observer's view. The left two images are color-coded so that red pixels show areas of the scene for which the shadow map is least magnified, and the blue pixels show areas where the shadow map is most magnified. In the left image, we see the scene from the light's viewpoint. The dark blue outline is the observer's view frustum.

In the middle image, we see the same scene, but from the observer's view. The yellow lines represent the light's view frustum, which faces the observer. Here, we see that areas of the image close to the viewer are precisely the areas where the shadow map has the greatest magnification when projected onto the image. Unfortunately, when using PSMs, the two perspective transforms (one from the light, one from the camera) mutually cancel and the result is that we're back to a standard, uniform shadow map.

The resulting shadow aliasing artifacts are seen in the right image.

In summary, PSMs are very useful, but they only address aliasing in certain cases. This is the main motivation for exploring another shadow algorithm, such as shadow silhouette maps.

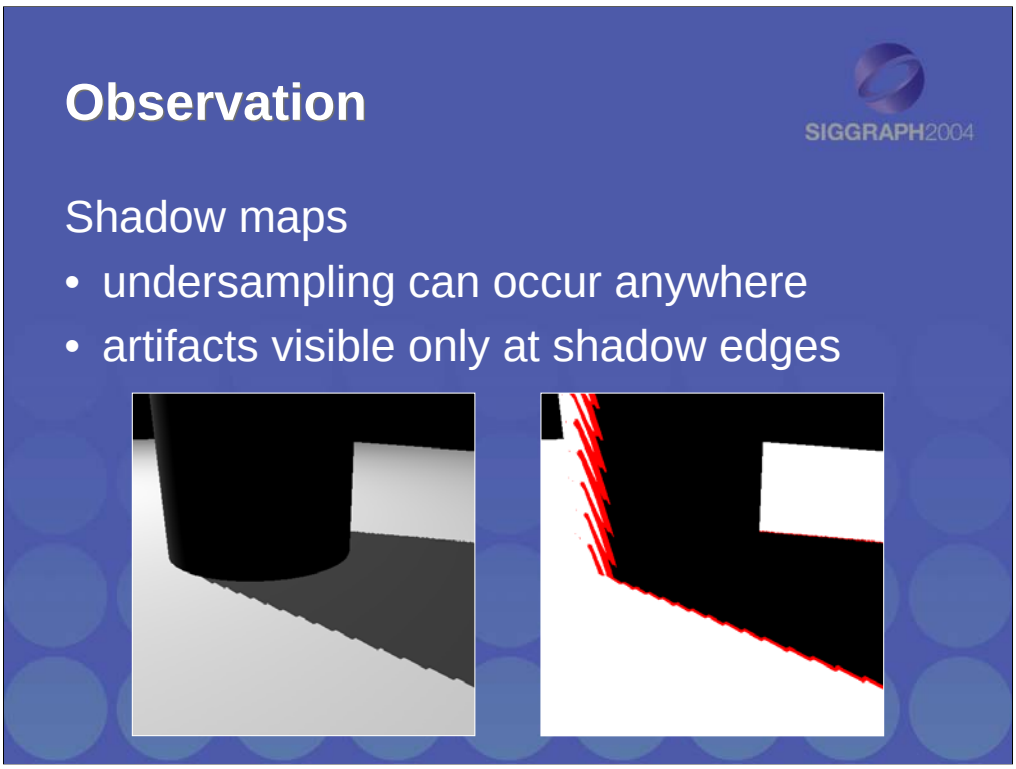
Shadow Silhouette Maps

- Research at Stanford University
  - P. Sen, M. Cammarano, and P. Hanrahan
  - Proceedings of SIGGRAPH 2003
- See course notes
- Also available online

The shadow silhouette map algorithm was developed at Stanford University by Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Their original paper which describes the method in detail was published in the Proceedings of ACM SIGGRAPH 2003. A copy of the paper should be included with these course notes. You can of course also download the paper online.

Pradeep Sen gave a nice talk at SIGGRAPH 2003, presenting the silhouette map algorithm. Many of the figures in these slides and notes are borrowed from his presentation.

Observation



The silhouette map algorithm is based on the following simple observation. Shadow maps can lead to undersampling, but the artifacts are visually objectionable only at the shadow silhouettes, i.e. the edges between shadowed and illuminated regions.

Observation

In contrast, shadow volumes give per-pixel accuracy everywhere, but this degree of accuracy is only needed at the silhouettes. The price for being accurate everywhere with shadow volumes is high fillrate and bandwidth consumption, illustrated in the right figure. The yellow polygons visualize shadow volume polygons. Clearly there is a lot of shadow volume overdraw in this scene, which leads to high fillrate and bandwidth. One of the characteristics of shadow maps, as we shall see, is relatively low bandwidth and fillrate consumption. This helps to keep the algorithm scalable to large scenes.

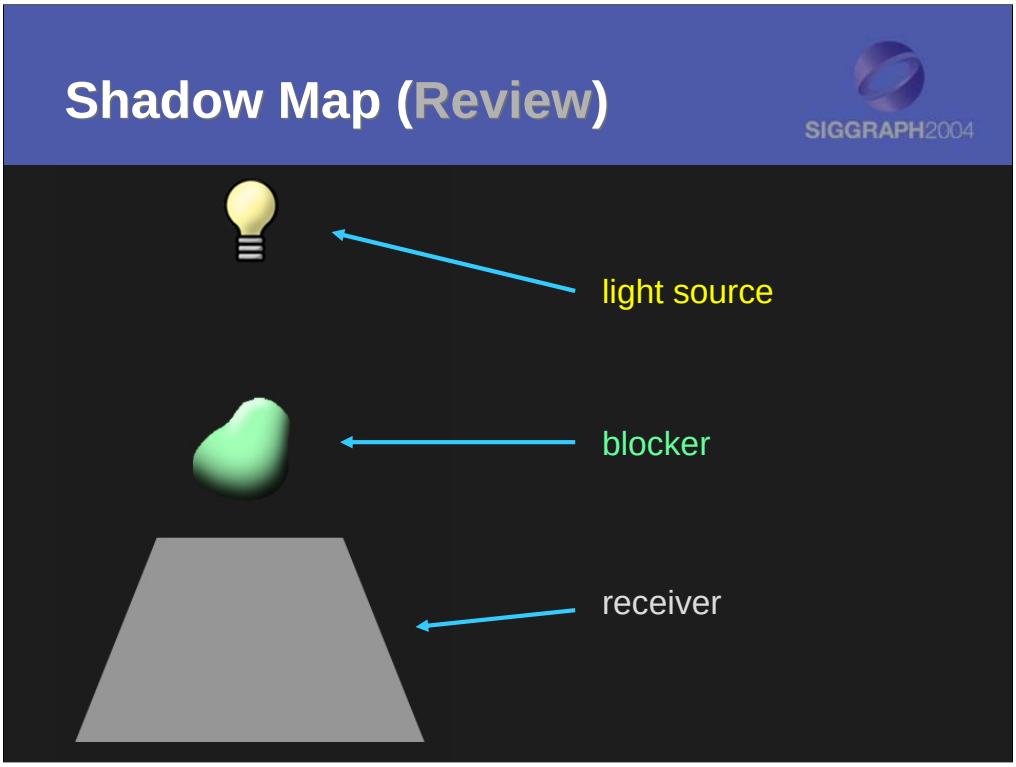
Algorithm Goals

- Accuracy of shadow volumes
- Efficiency of shadow maps
- Treats perspective and projection aliasing
- Supports dynamic scenes
- Maps to graphics hardware

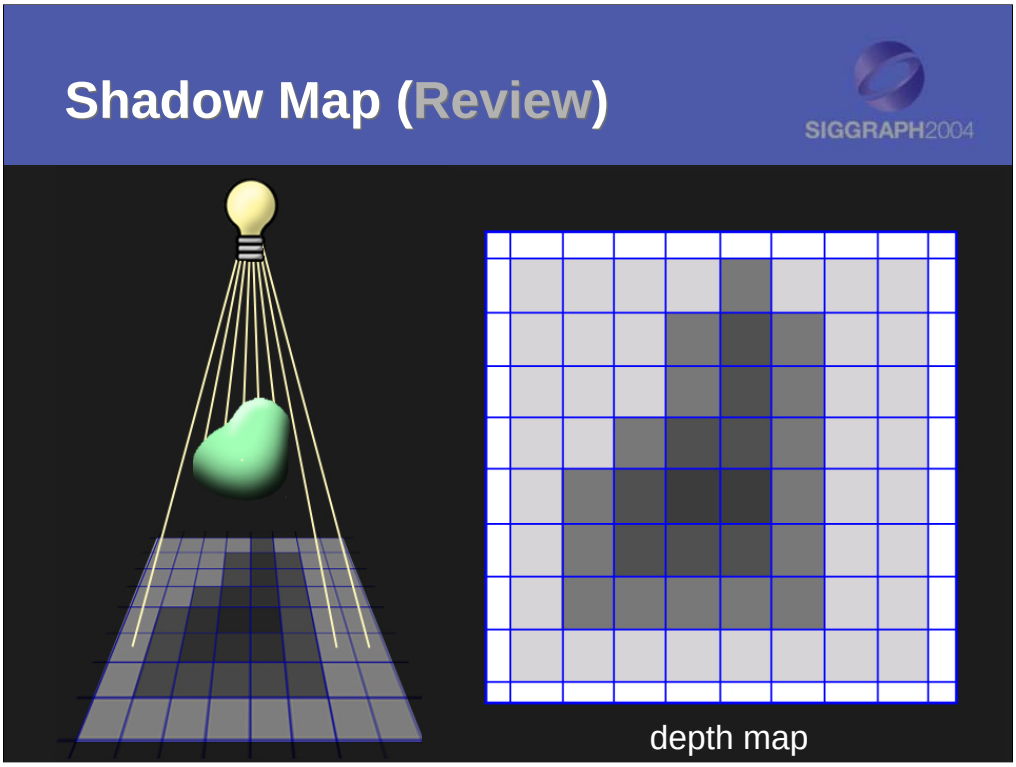
We want a hybrid algorithm that combines the best characteristics of shadow maps and shadow volumes. The silhouette map algorithm will focus on the shadow silhouettes, since those are the pixels in the image that are critical to get right.

These are the goals of the silhouette map algorithm. Ideally, we would have the accuracy of shadow volumes and the efficiency of shadow maps. As we'll see, silhouette maps don't quite achieve this goal, but they do offer an excellent tradeoff. Silhouette maps are designed to work on dynamic scenes, i.e. the light, observer, and objects can move from frame to frame; no precomputation is required. Finally, silhouette maps are designed to be simple enough to implement on graphics hardware. As we'll see, though, they make heavy use of the programmable features of modern graphics hardware.

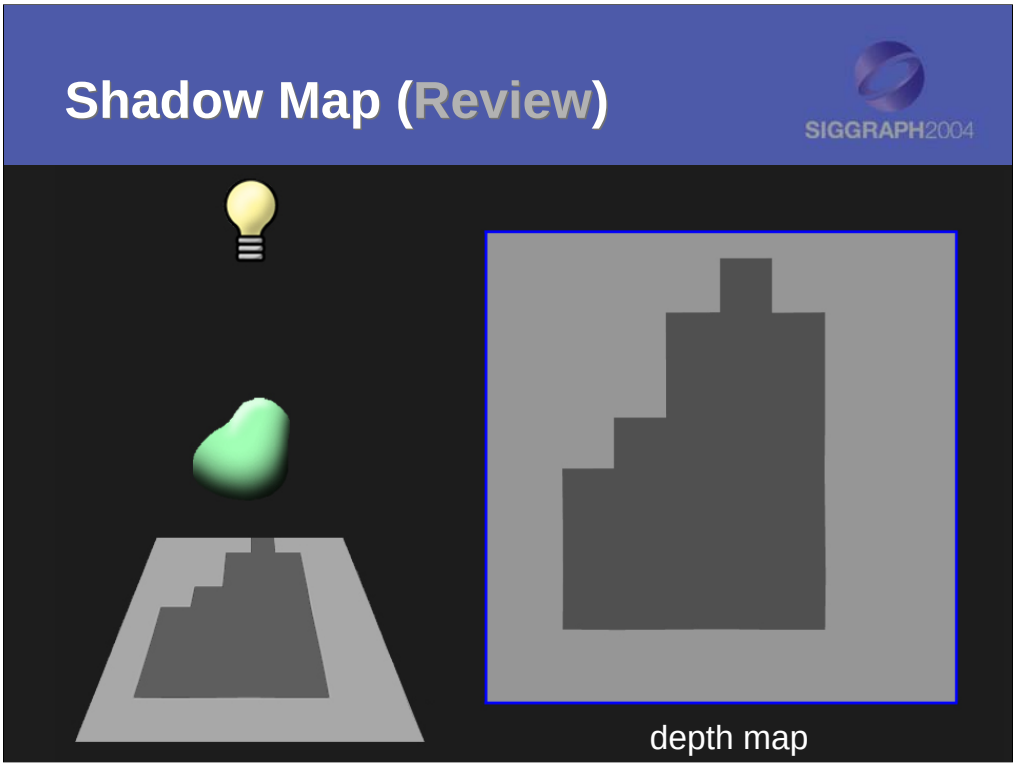
The basic idea is to augment the normal depth map with an extra buffer, called a silhouette map, which helps to represent shadow silhouettes more accurately.



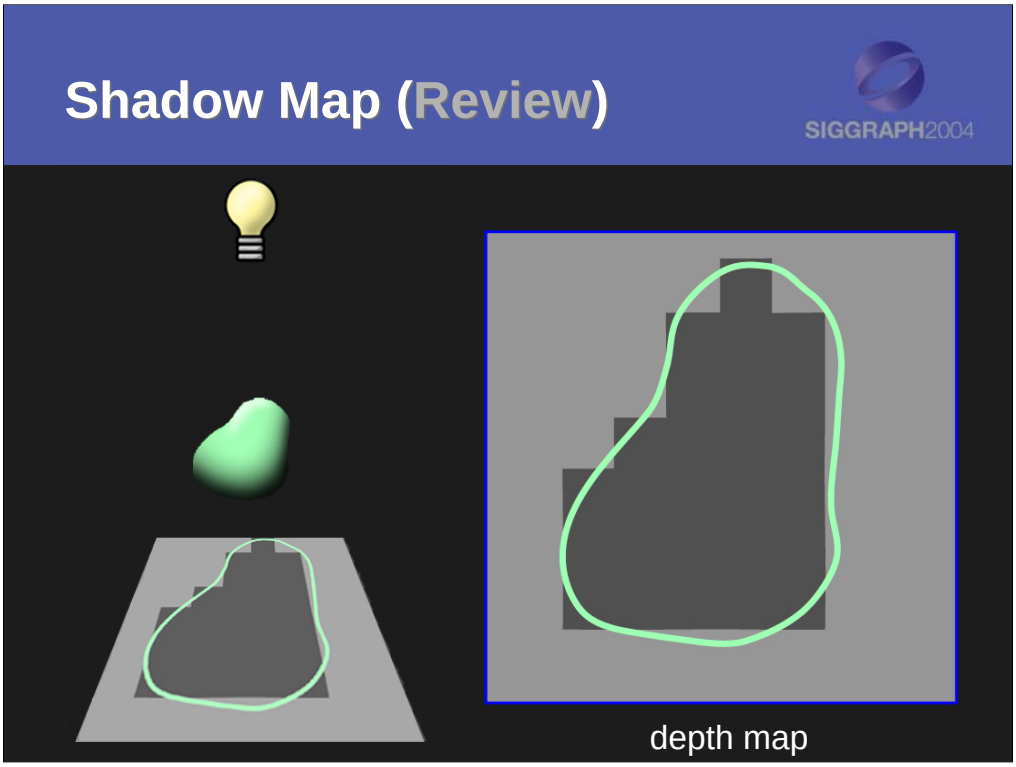
To understand how the silhouette map algorithm works, let's review how the regular shadow map algorithm works. We have a light source, blocker, and receiver.



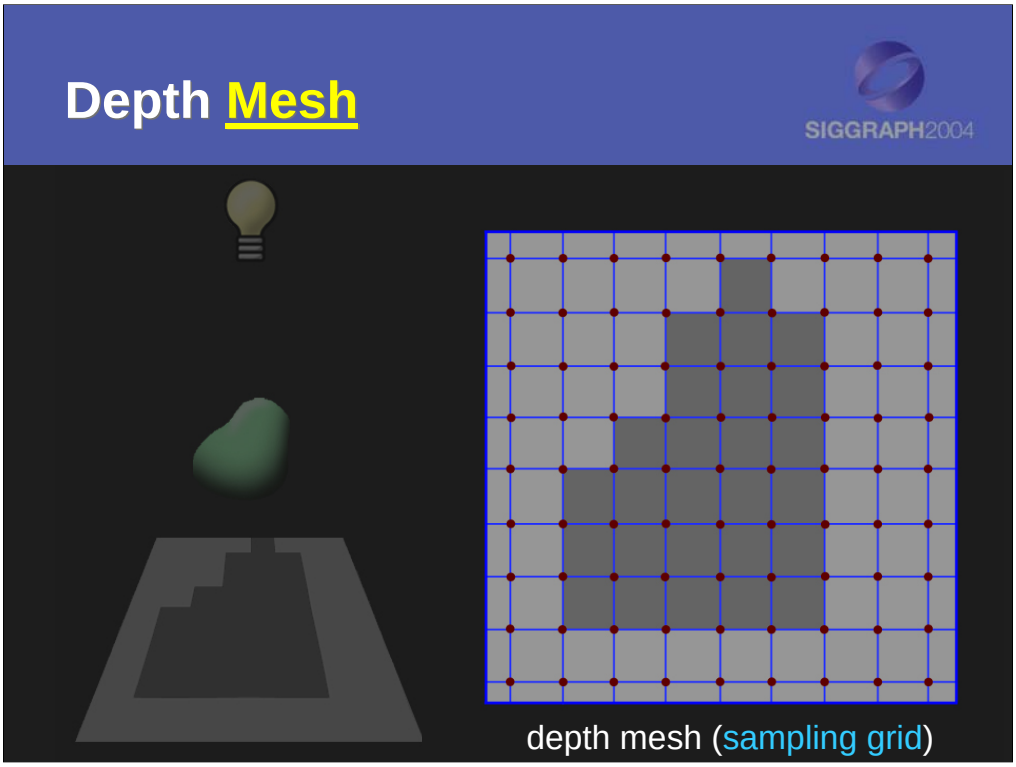
We rasterize the blocker into a depth map. In the visualization on the right, dark values represent small depths (close to the light), and light values represent large depths (far from light).



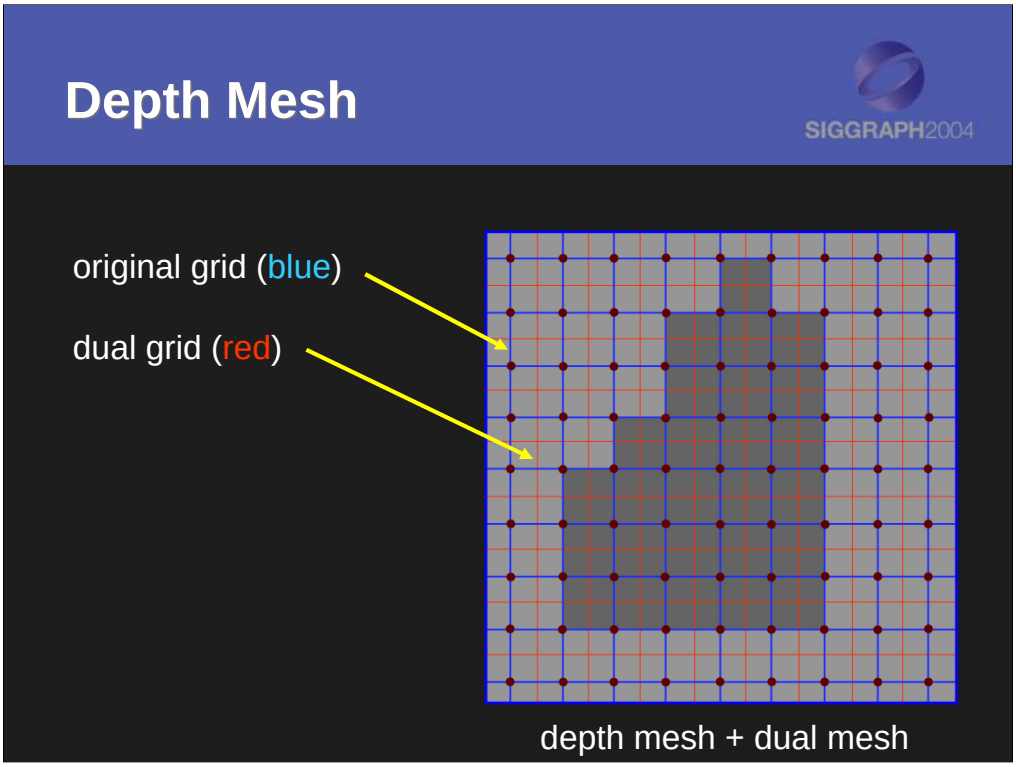
Due to limited depth buffer resolution, we obtain a poor representation of the blocker's shadow silhouette.



For reference, the true silhouette curve is shown in green. We can think of the standard shadow map algorithm as providing a piecewise-constant approximation to the shadow contour. This is because all samples in the final image that get mapped to a given texel in the shadow map will have the same binary value: 0 if the depth test fails, 1 if the depth test passes. It is this piecewise-constant approximation that leads to blocky aliasing artifacts in the final image.

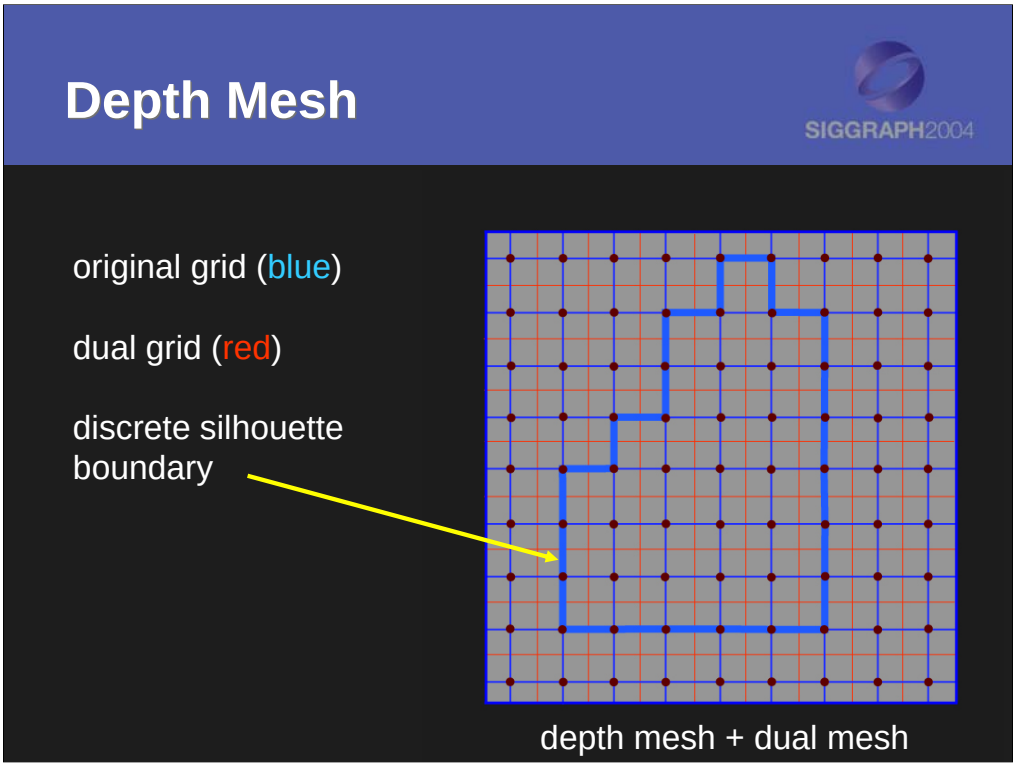


Now let's see how we can improve the approximation. Let's stretch our minds a bit and think of the depth map not as a buffer, but as a mesh where each sample is a vertex of the mesh. In 2D, the mesh is just a uniform, rectilinear grid as shown here.

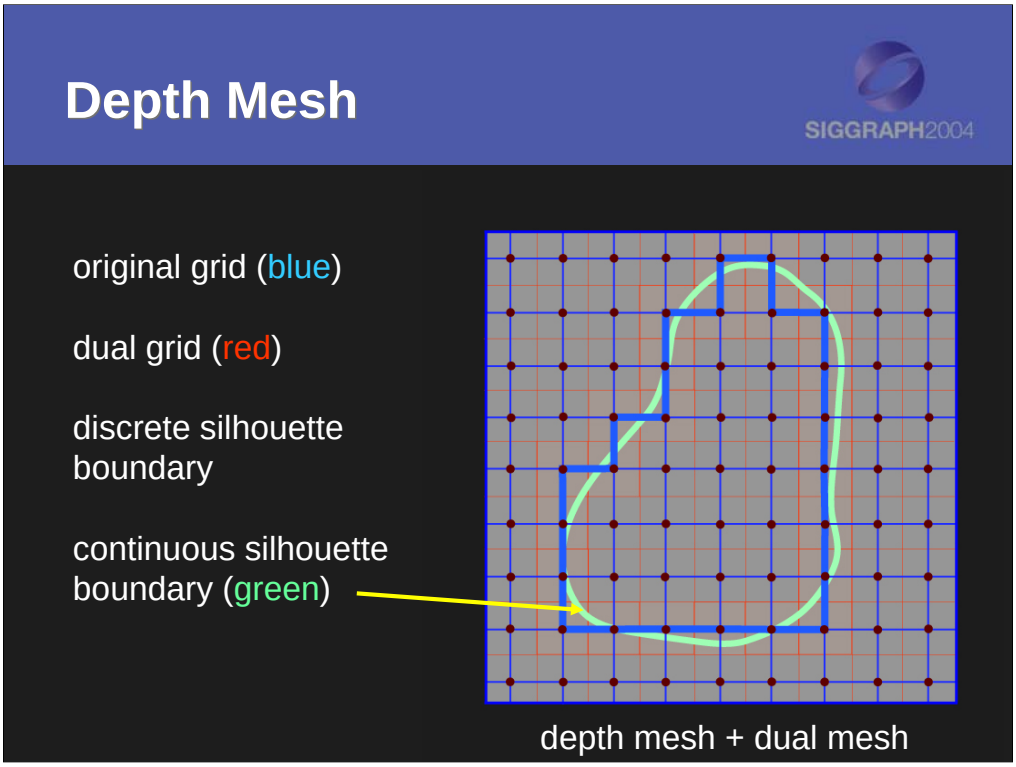


The depth mesh is shown in blue. As we'll soon see, it will be useful also to consider the dual mesh, shown in red. Practically, this is the same as the original grid, but offset by ½ a pixel in both x and y.

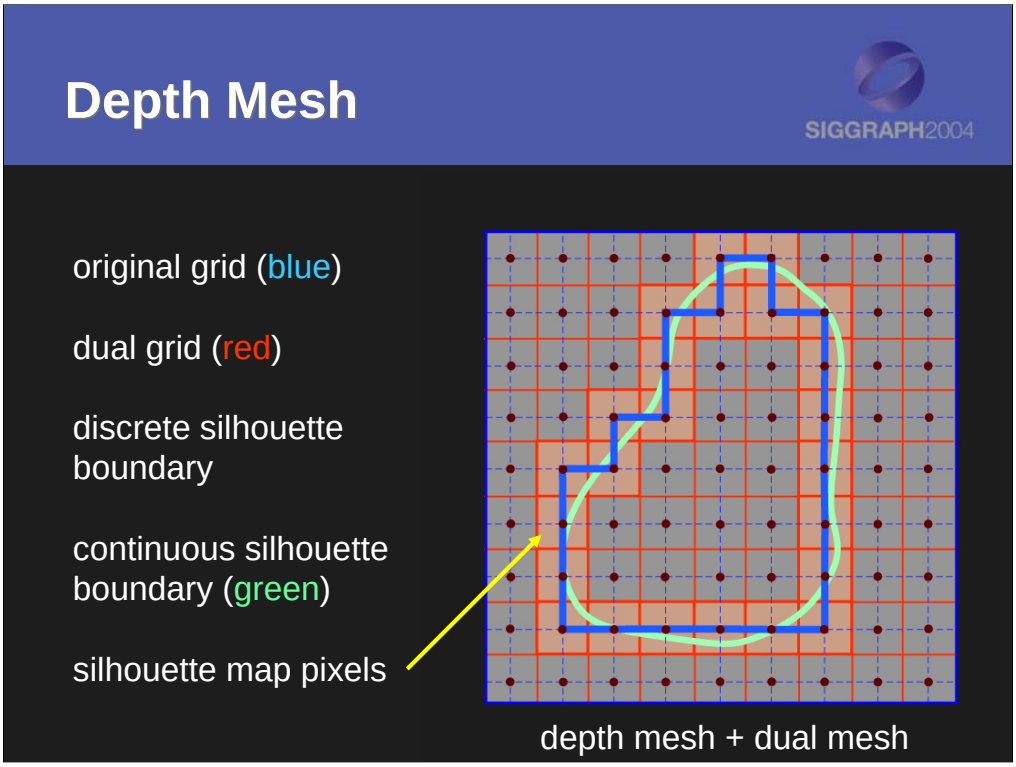
Depth Mesh



With the standard shadow map algorithm, we get discrete shadow boundaries that are aligned with the rectilinear grid of the depth mesh.

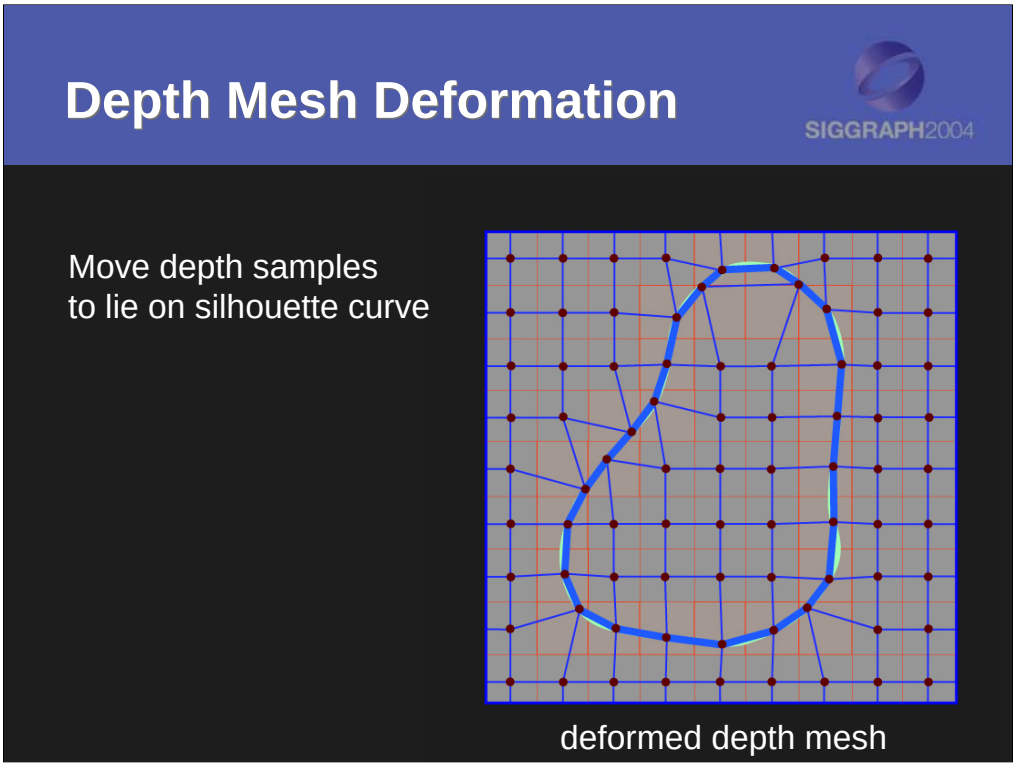


Here's the magic: why restrict ourselves to a regular depth mesh? It would be better if we could somehow deform the depth mesh so that the samples are better aligned with the true silhouette boundary (shown in green). In fact, we can do just that.

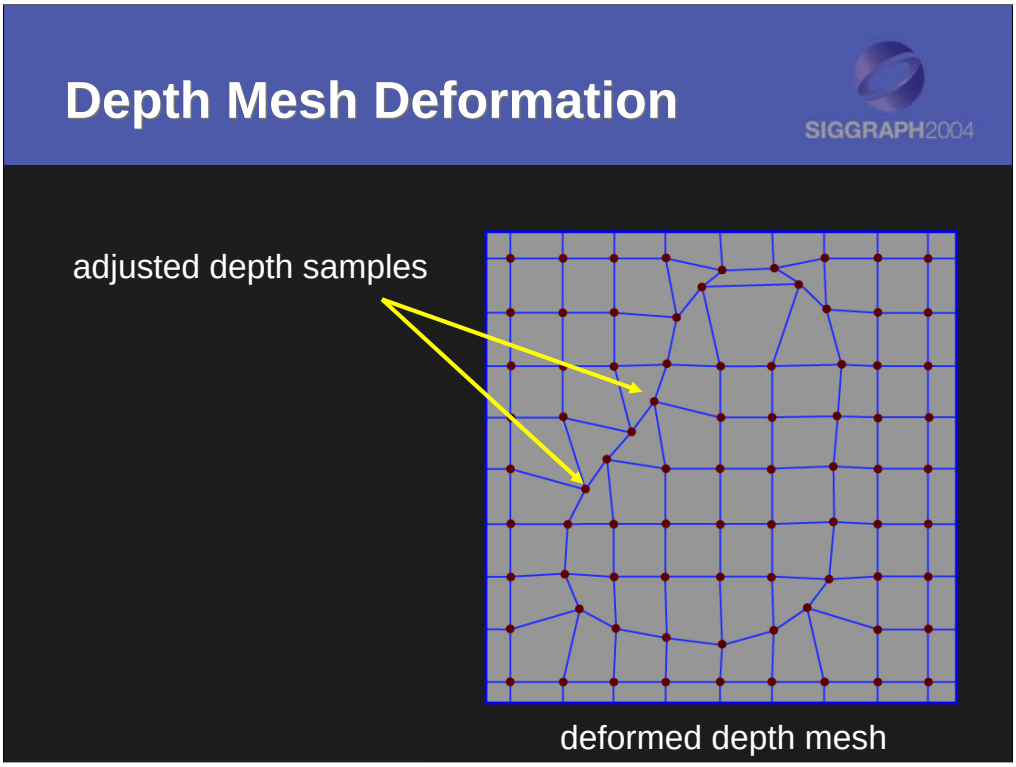


Here's where the dual mesh comes in handy. Look at all the cells of the dual grid (highlighted in red) that contain the continuous silhouette curve. These are also the cells that contain the discrete, grid-aligned shadow boundary (dark blue).

Depth Mesh Deformation

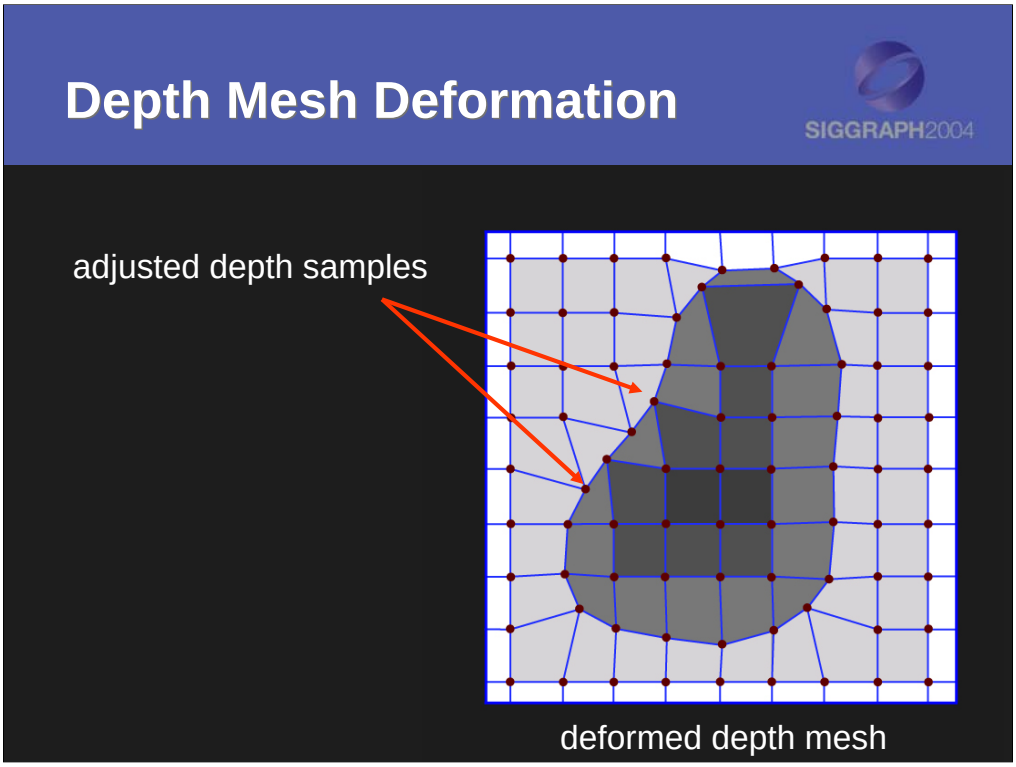


The idea is to deform the depth mesh: move the relevant depth samples so that they lie precisely on the silhouette curve itself. This results in the deformed depth mesh shown here.

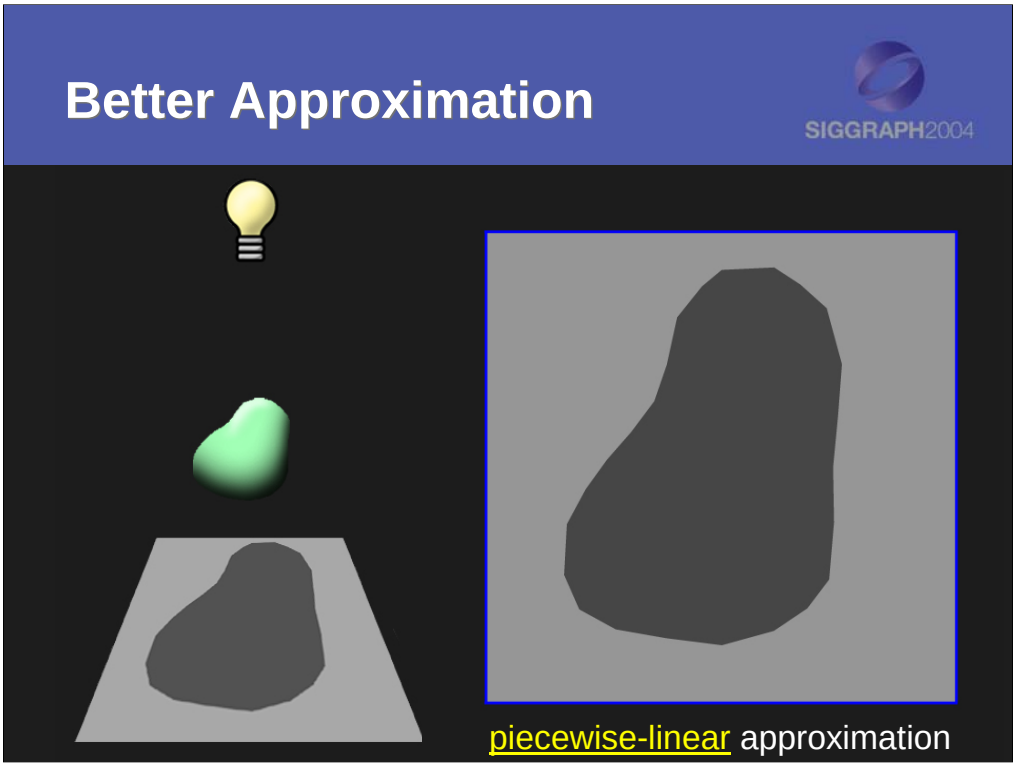


We still have the same number of depth samples as before. It just so happens that some of them have been moved from their original position to lie on the silhouette curve.

Depth Mesh Deformation

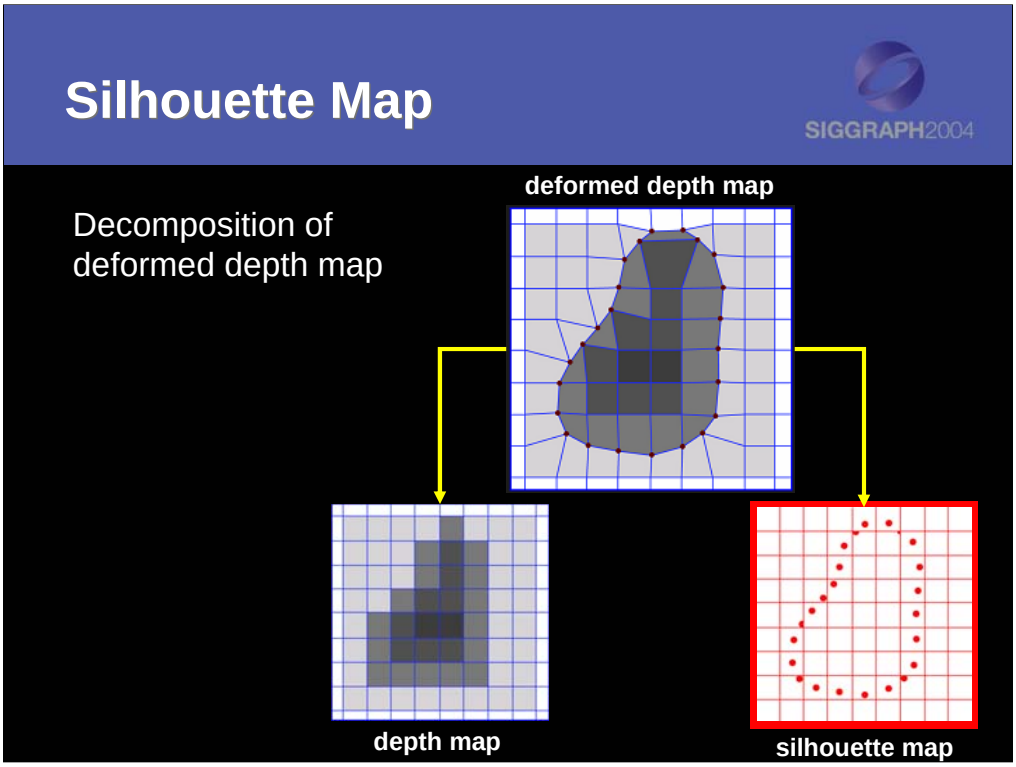


Now let’s see when we apply the shadow map algorithm, this time with the deformed mesh.



Notice that now we get a much better approximation to the shadow silhouette. Instead of a piecewise-constant approximation to the contour, we have piecewise-linear approximation (imagine connecting the dots of the deformed depth samples).

Conceptually, this is all there is to the silhouette map algorithm.



In practice, however, we can't easily create or use deformed depth meshes on graphics hardware. But we can look at the problem from a slightly different angle. Instead of working directly with a deformed depth mesh, we can use two buffers: a regular depth map and a silhouette map, a 2D image that we'll describe in a moment. Together, these buffers effectively give you a deformed depth map that can be used to give the piecewise-linear approximation of the shadow boundary.

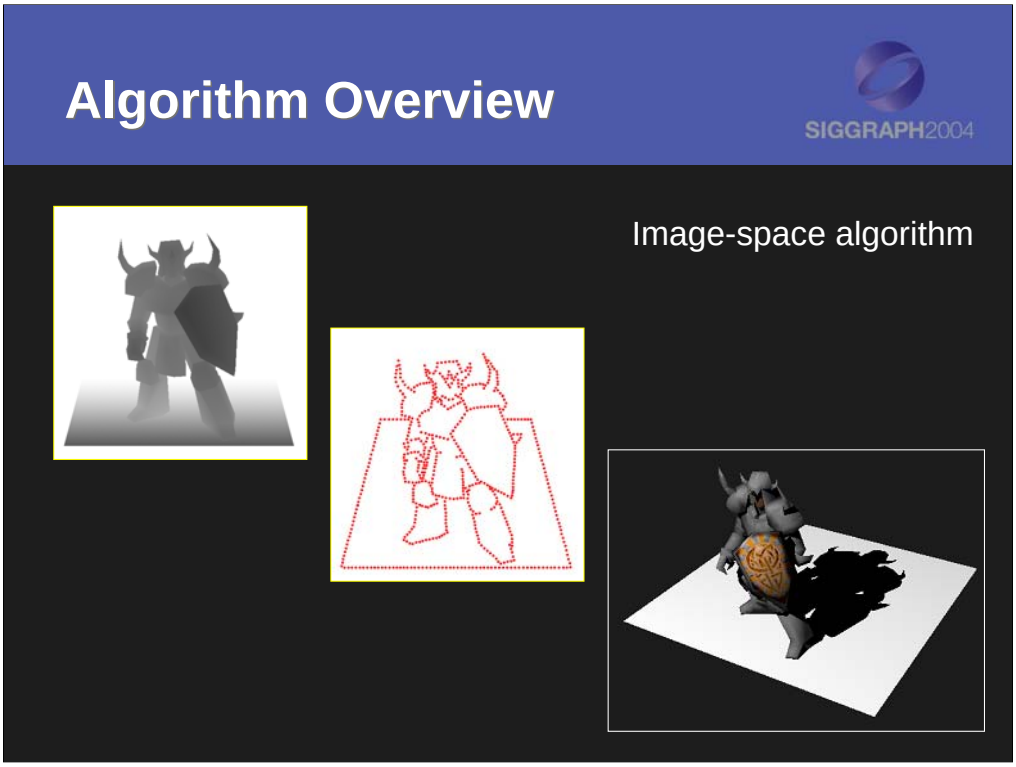
What is a Silhouette Map?

Many ways to think about it:

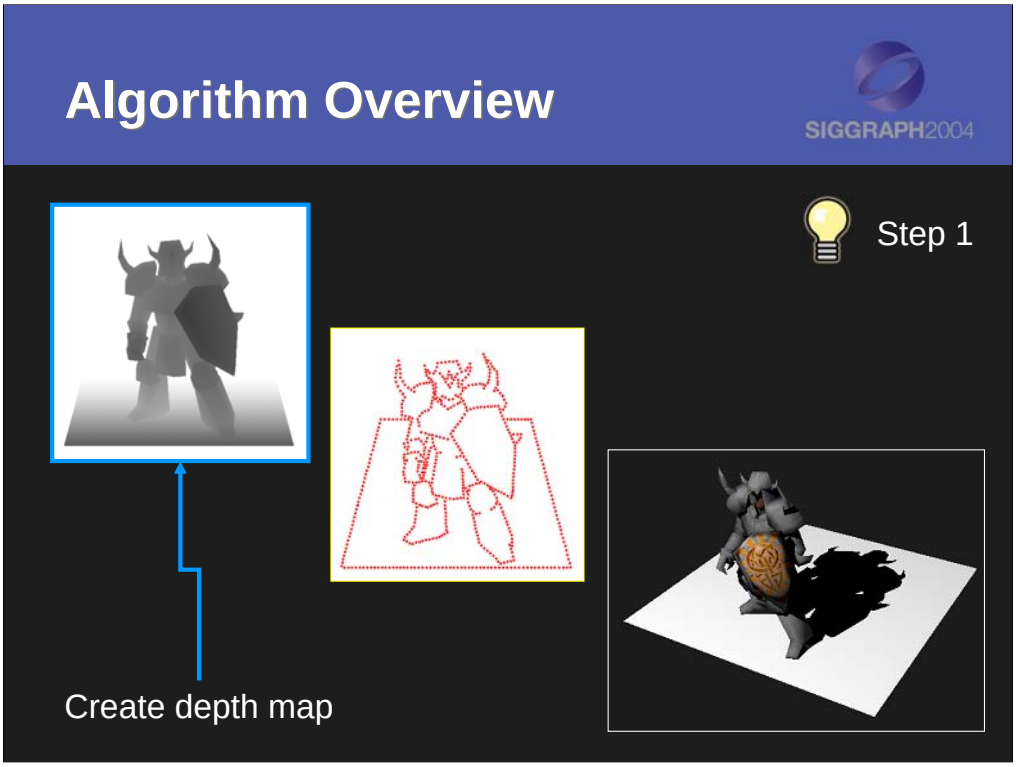
- Edge representation
- 2D image, same resolution as depth map
- Offset from depth map by ½ pixel in x, y
- Stores xy-coordinates of silhouette points
- Stores only one silhouette point per texel
- Piecewise-linear approximation

So what exactly IS a silhouette map? There are many ways to think about it. Abstractly, it's an edge representation, meaning that its main purpose is to store accurate edge information. After all, the overall goal here is to improve the shadow silhouette. Concretely, the silhouette map is just a buffer offset from the depth map by ½ pixel in both x and y. Each texel in the silhouette map stores a single point; for texels that are crossed by the silhouette curve, the texel stores a point that lies on the curve. Note that only one silhouette point is stored per texel. We'll see some implications of this restriction.

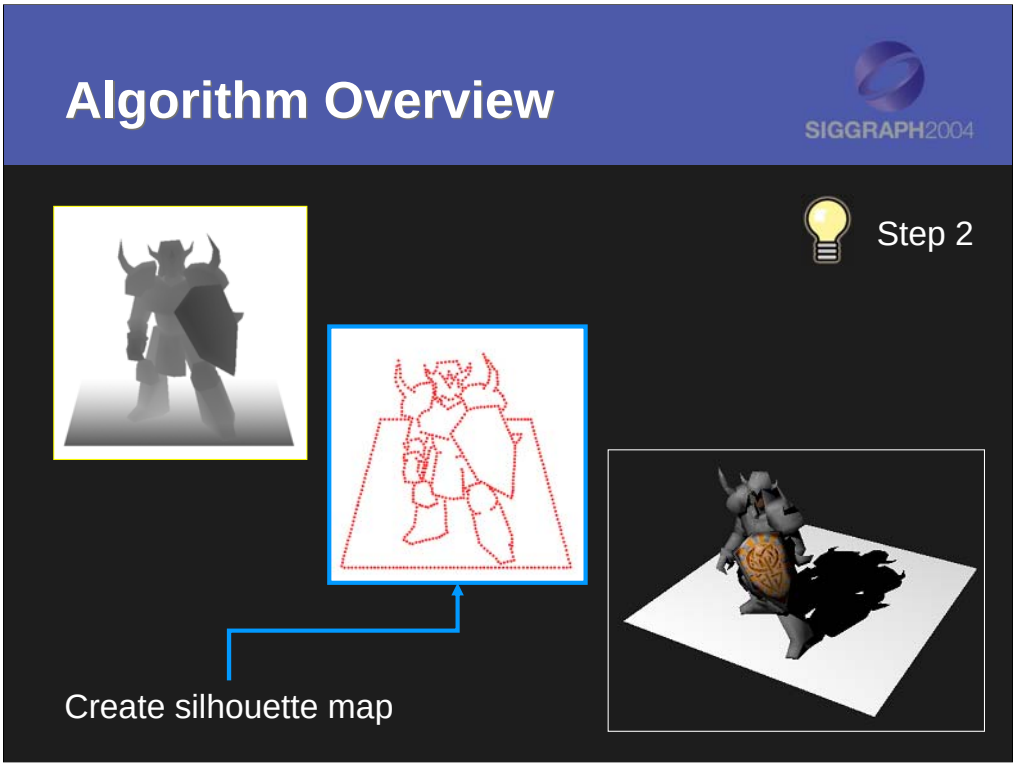

SIGGRAPH2004
Algorithm



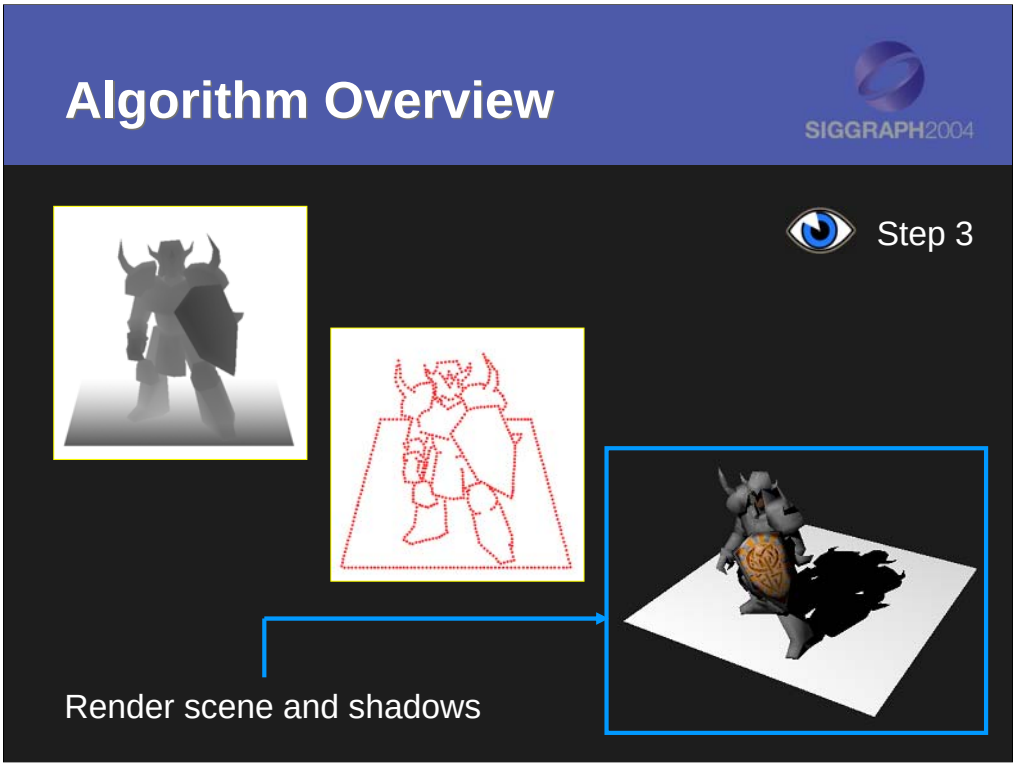
Here's an overview of the algorithm. There are 3 rendering passes.



The first pass draws a regular depth map from the light's viewpoint. This is exactly the same as in the standard shadow map algorithm.

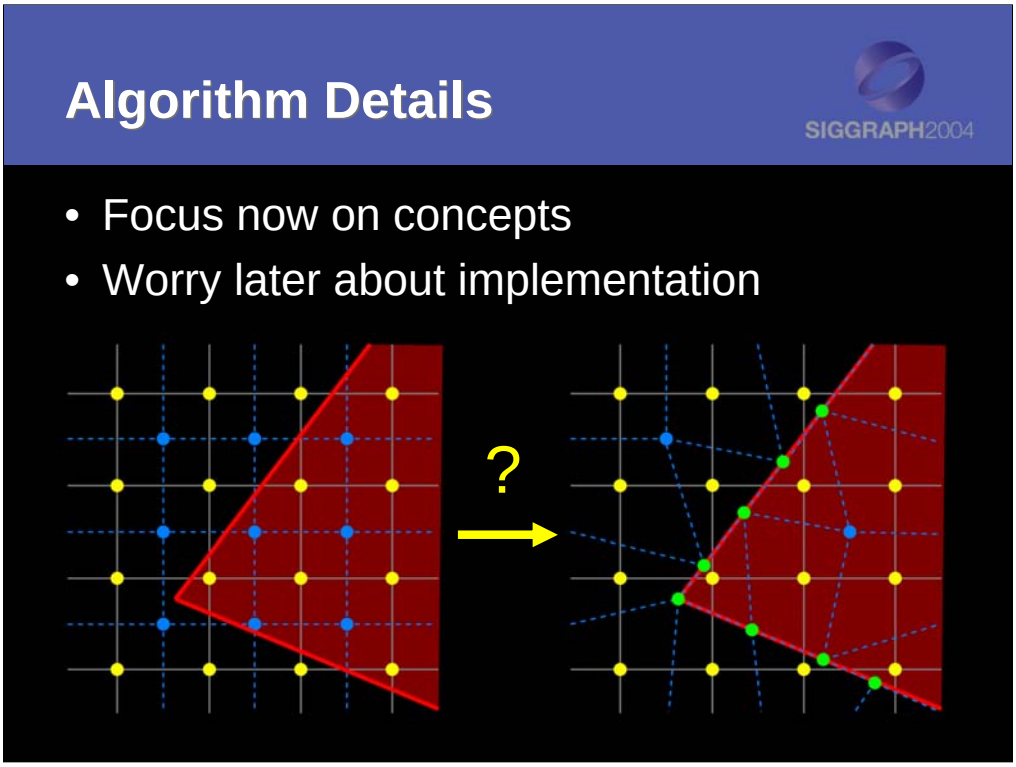


The second step is to render a silhouette map, also from the light's viewpoint. We'll see later exactly how this is done.



Finally, we render the scene from the observer's viewpoint and refer to both the shadow map and silhouette map to obtain accurate shadows.

Algorithm Details



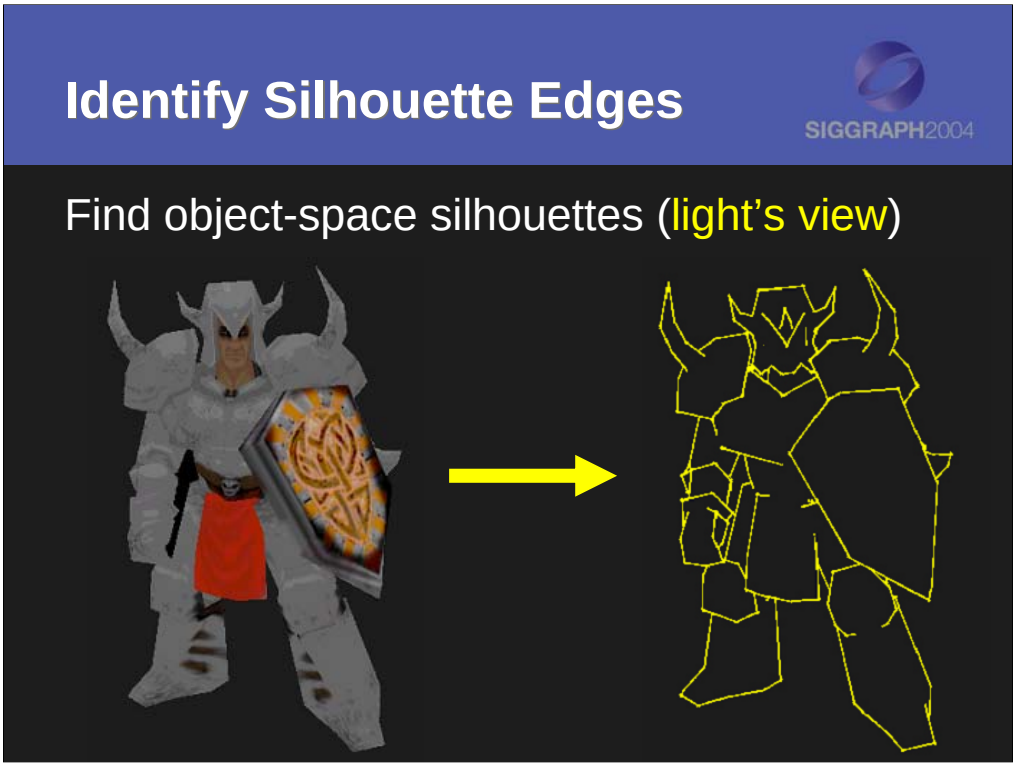
This part of the discussion will focus on the concepts of the algorithm. It may not yet be clear how to implement this in hardware. Don't worry; we'll cover this later.

Create Depth Map

Same as in regular shadow maps

The first step is to render a depth map.

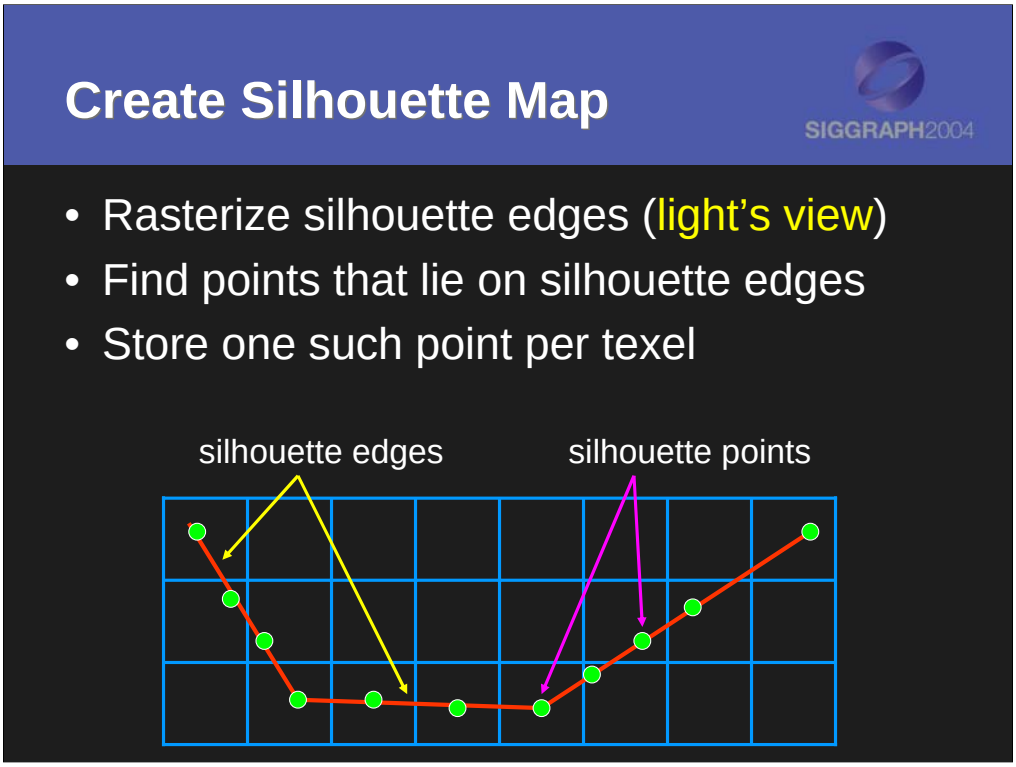
Identify Silhouette Edges



Before we can create a silhouette map, we have to identify the silhouette edges. This task is sometimes referred to as object-space silhouette extraction. For polygonal models, a simple way to perform this task is to loop through all edges and check to see if one of its adjacent faces is facing the light and another is facing away. This approach may sound overly simplistic, but in fact it's one of the best methods available for dynamic scenes with animated characters. Best of all, it's easy to implement and always works.

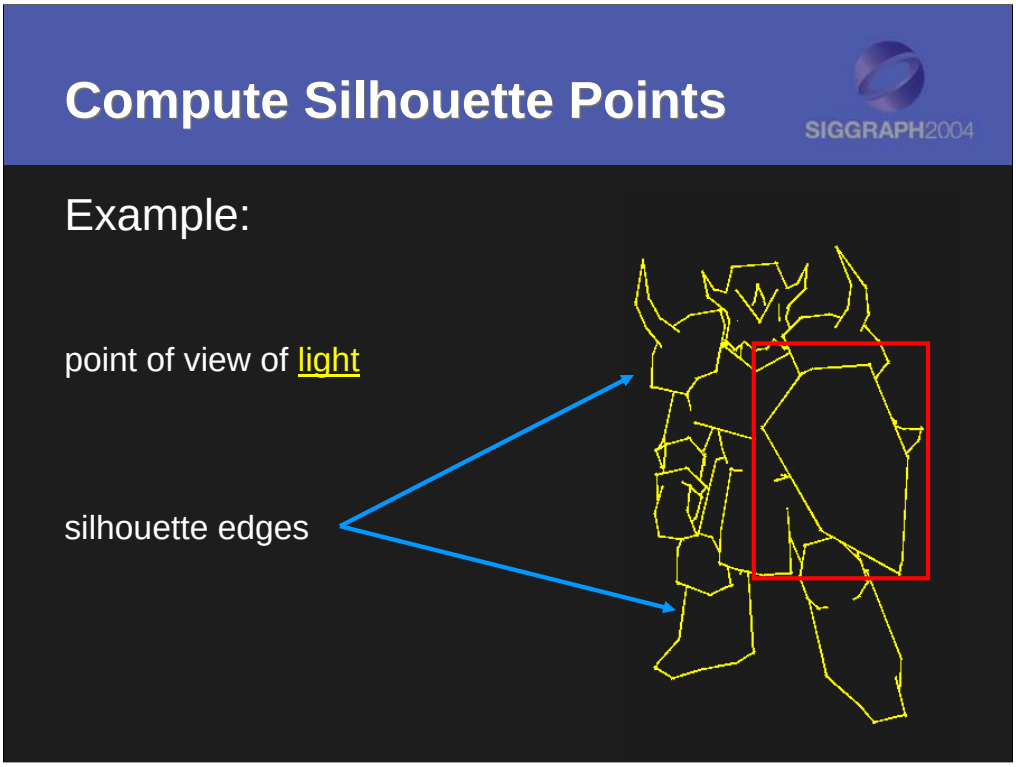
Note the assumption that we've made here: objects (in particular, the blockers in the scene) are represented as polygons.

Create Silhouette Map

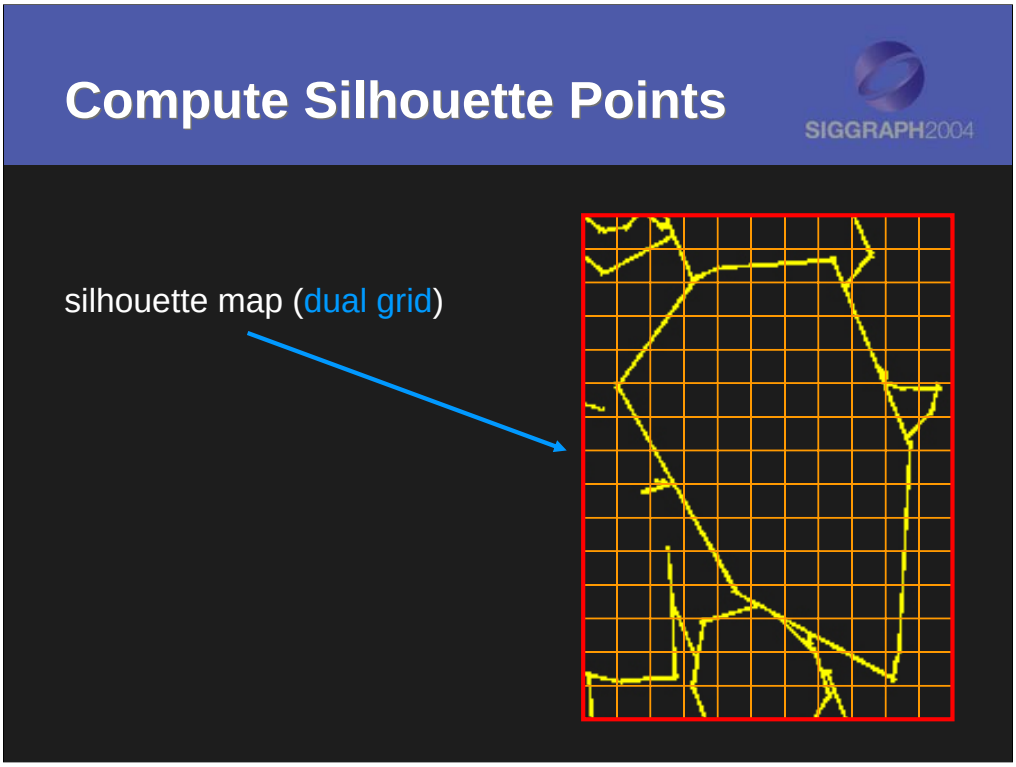


Now that we have the silhouette edges, we draw them (again from the light's viewpoint) to generate the silhouette map. The overall idea is to pick points that lie on the edges and store these points in the silhouette map.

Compute Silhouette Points

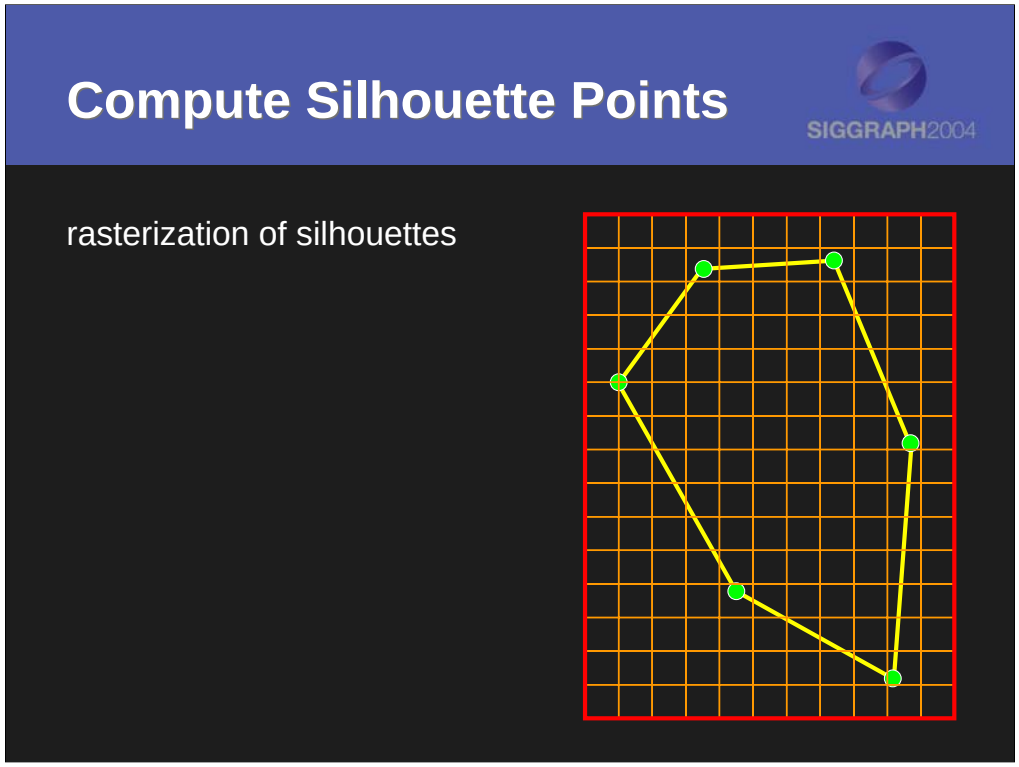


Let's use an example to understand exactly how this process works. Here is a visualization of the silhouette edges of a Knight character, seen from the point of view of the light source. We'll focus on the part outlined in red, i.e. the Knight's shield.



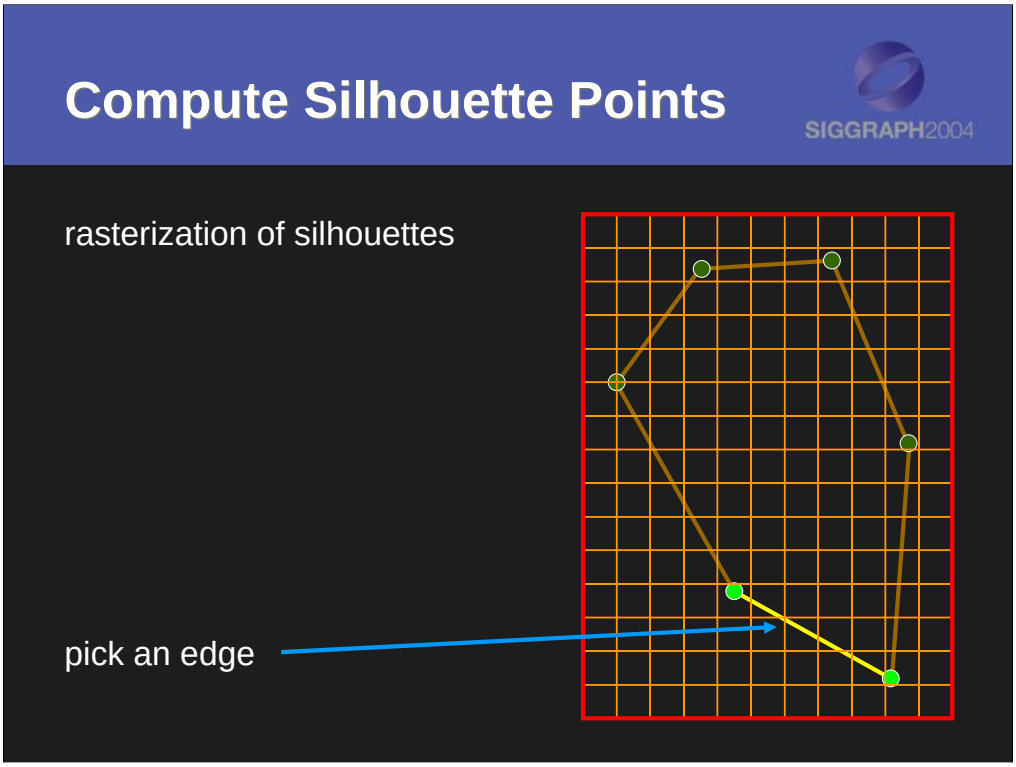
The grid lines in this image show the pixel boundaries of the silhouette map. Remember that the silhouette map is the dual to the original depth map, i.e. it's offset by ½ pixel in both x and y.

Compute Silhouette Points



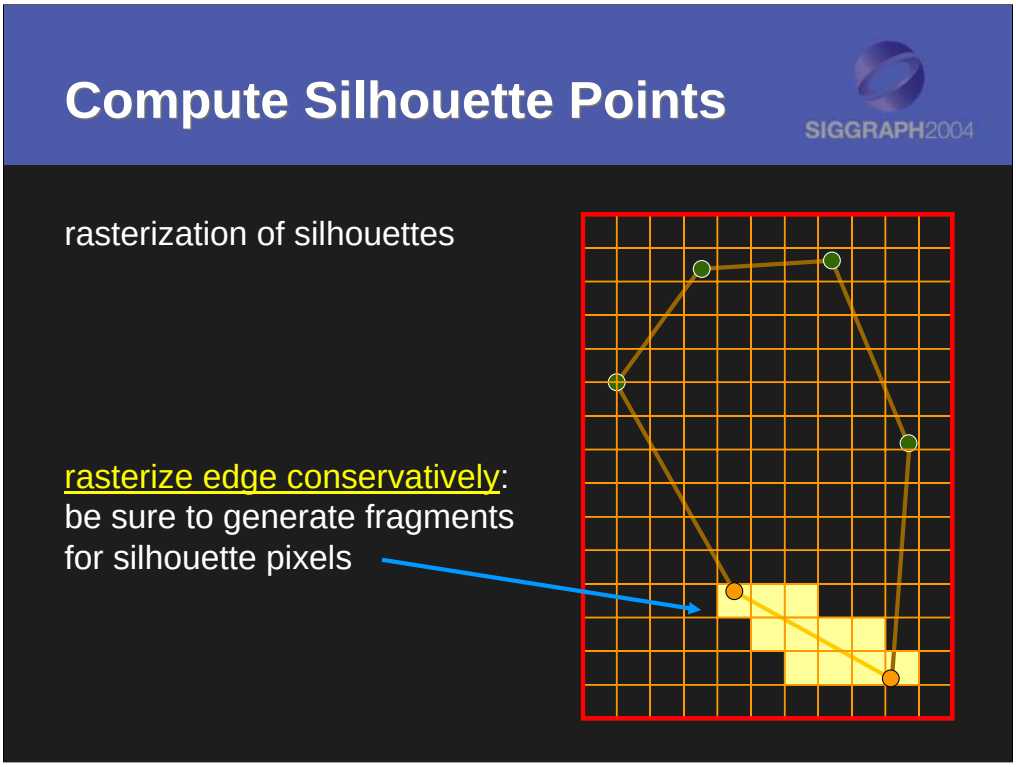
Now let’s see how we rasterize the silhouette edges and wind up with silhouette points.

Compute Silhouette Points



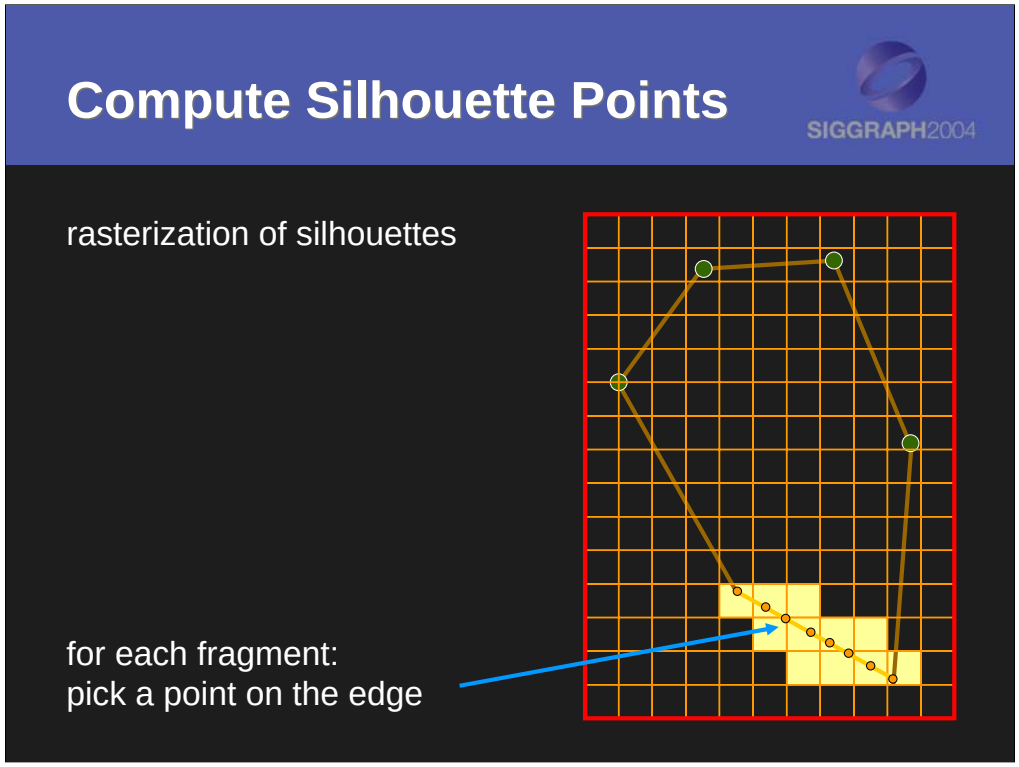
Pick any edge to start with.

Compute Silhouette Points



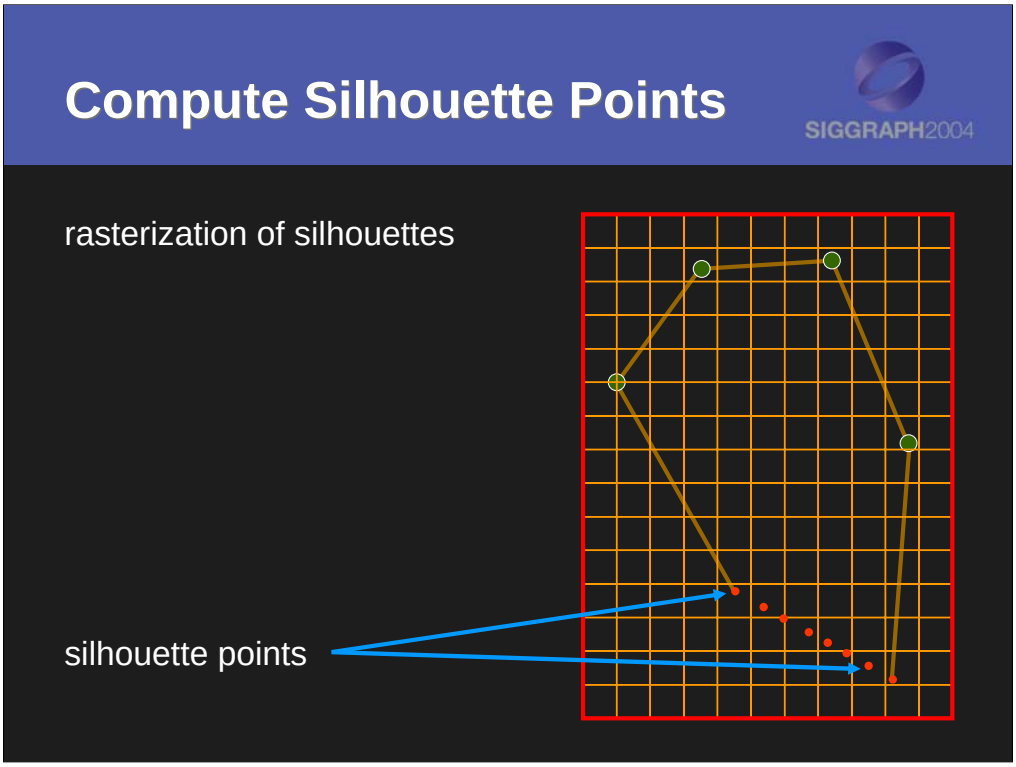
We rasterize the edge conservatively, meaning that we must guarantee that all fragments (i.e. pixels) that are crossed by the silhouette edge are rasterized. (We'll see later how to guarantee this.) The generated fragments are highlighted above. Note that since we are rasterizing conservatively, it is possible that some fragments will be generated that are not crossed by any silhouette edge. Again, later we'll see how to handle this situation.

Compute Silhouette Points



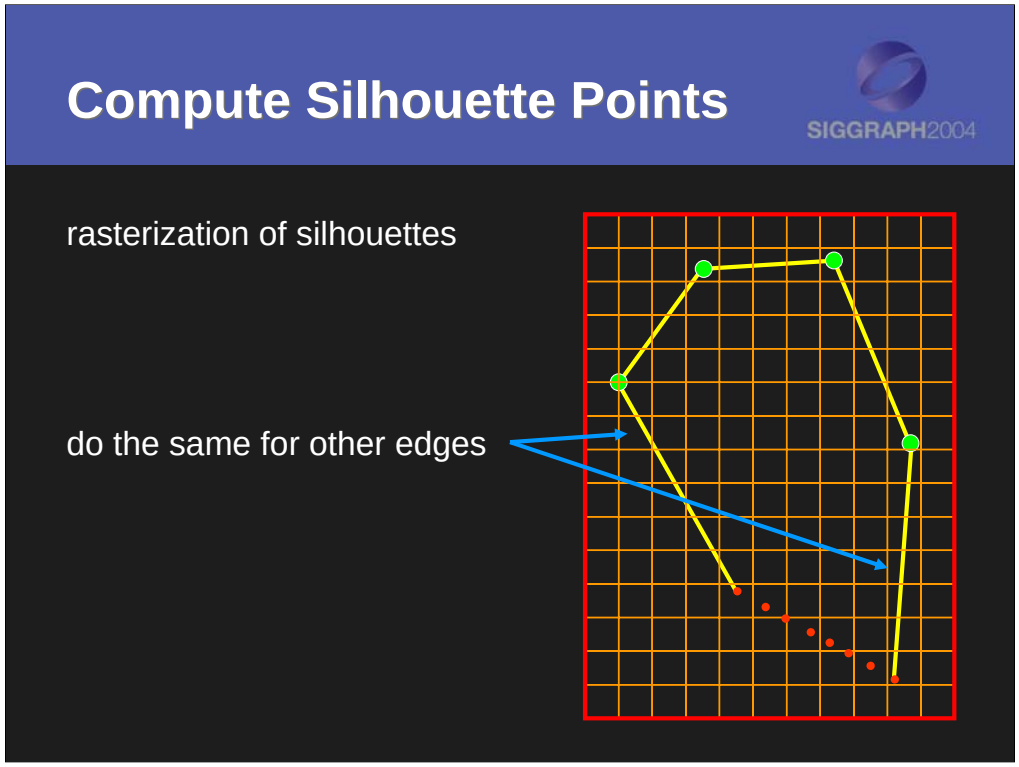
Each fragment generated by the rasterizer will eventually end up in a texel in the silhouette map. For each fragment, we pick a point that lies on the silhouette edge.

Compute Silhouette Points



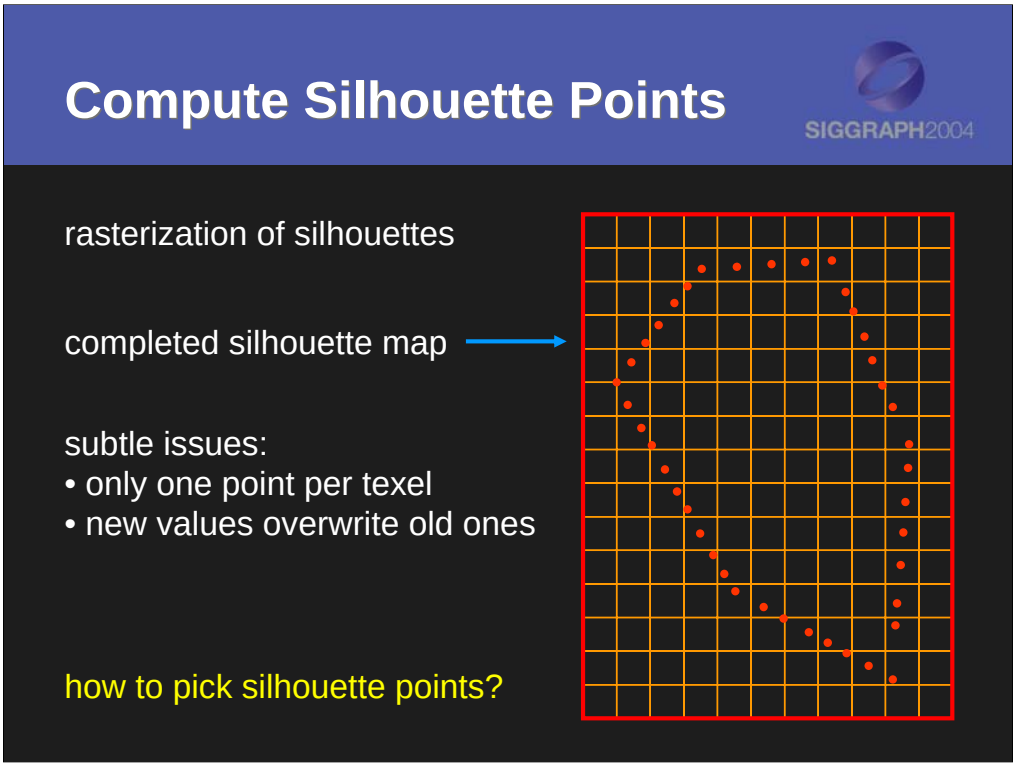
We store the coordinates of these points into the silhouette map.

Compute Silhouette Points



Repeat this step for all the silhouette edges.

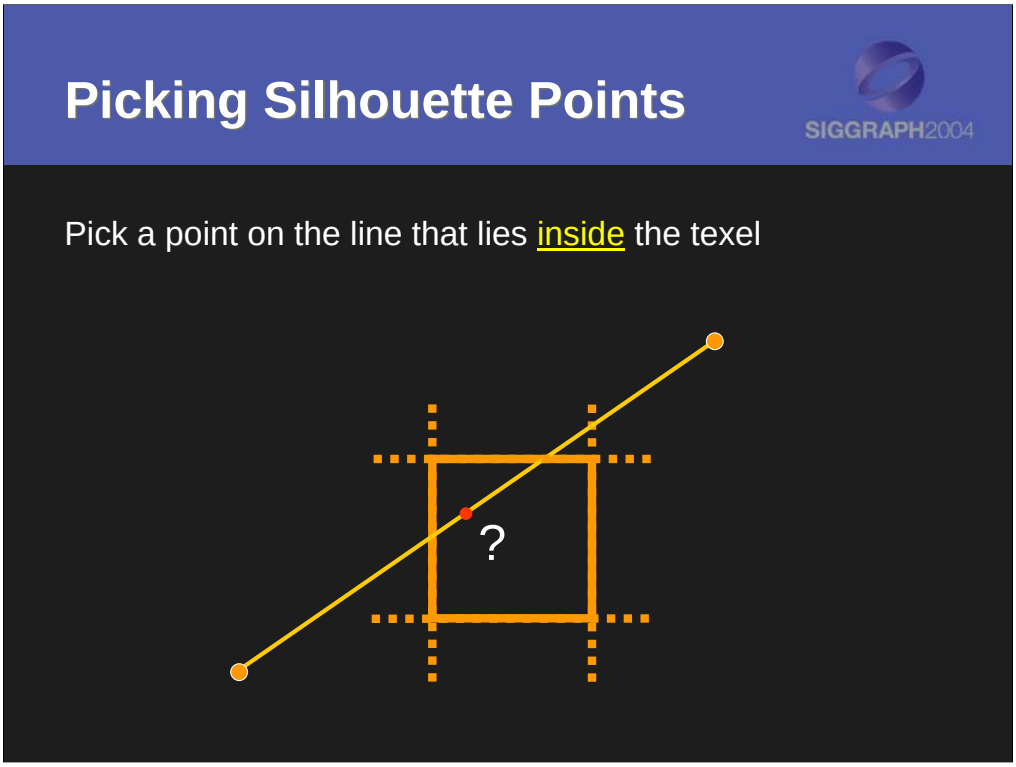
Compute Silhouette Points



Here is a visualization of what the final silhouette map might look like after rasterizing all the silhouette edges and storing the associated silhouette points into the buffer. There are two issues to be aware of. Remember that we only store one point per texel in the silhouette map. (The reason for storing just one point is that it makes the algorithm consistent and easier to implement on graphics hardware. The alternative, storing multiple points per texel, complicates matters.) The second issue is that multiple edges may cross a single texel of the silhouette map. Since only one value may be stored per texel, we (somewhat arbitrarily) choose to let new silhouette points overwrite old ones. Thus only the last silhouette point written to a texel will be kept.

Now that we've seen the overall approach, the main question is: how do we pick the silhouette points?

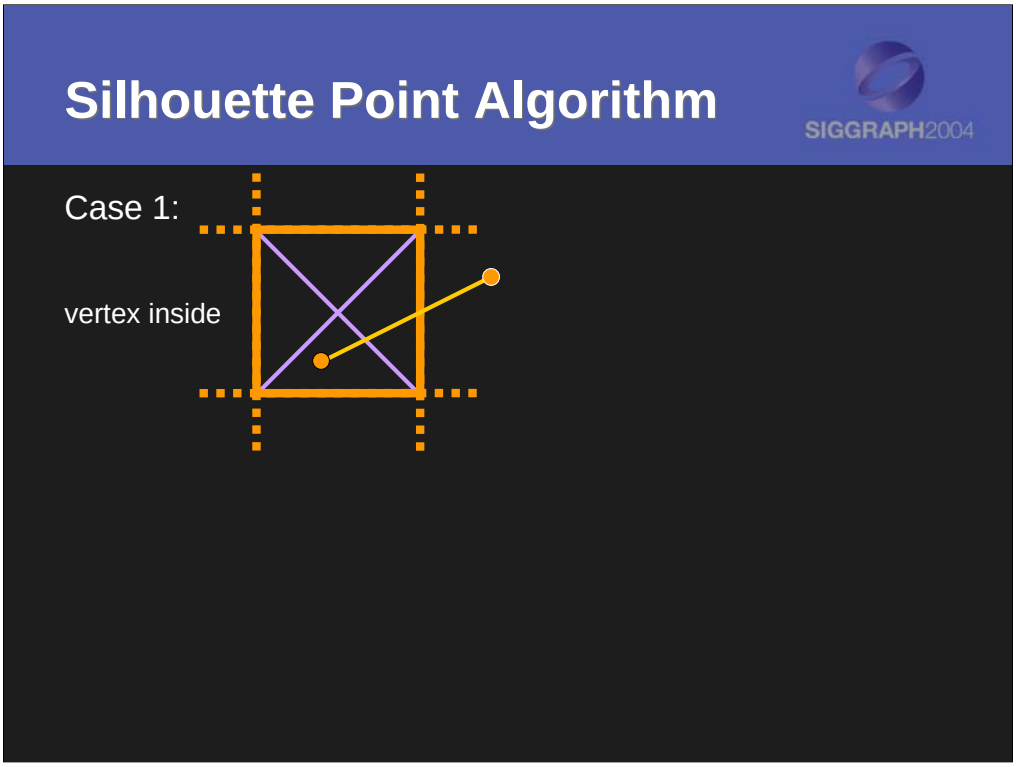
Picking Silhouette Points



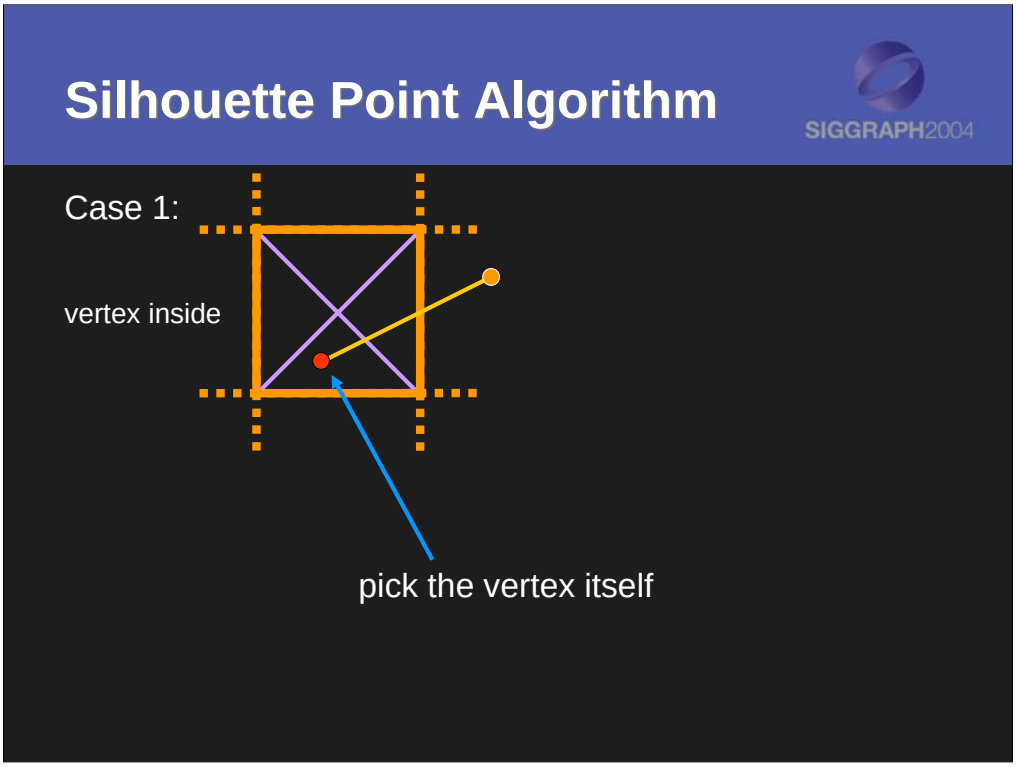
Let's say we've drawn a silhouette edge, and the rasterizer has generated a bunch of fragments. For each fragment covered by the edge, we want to pick a point.

We'll break the problem of picking points into a few cases.

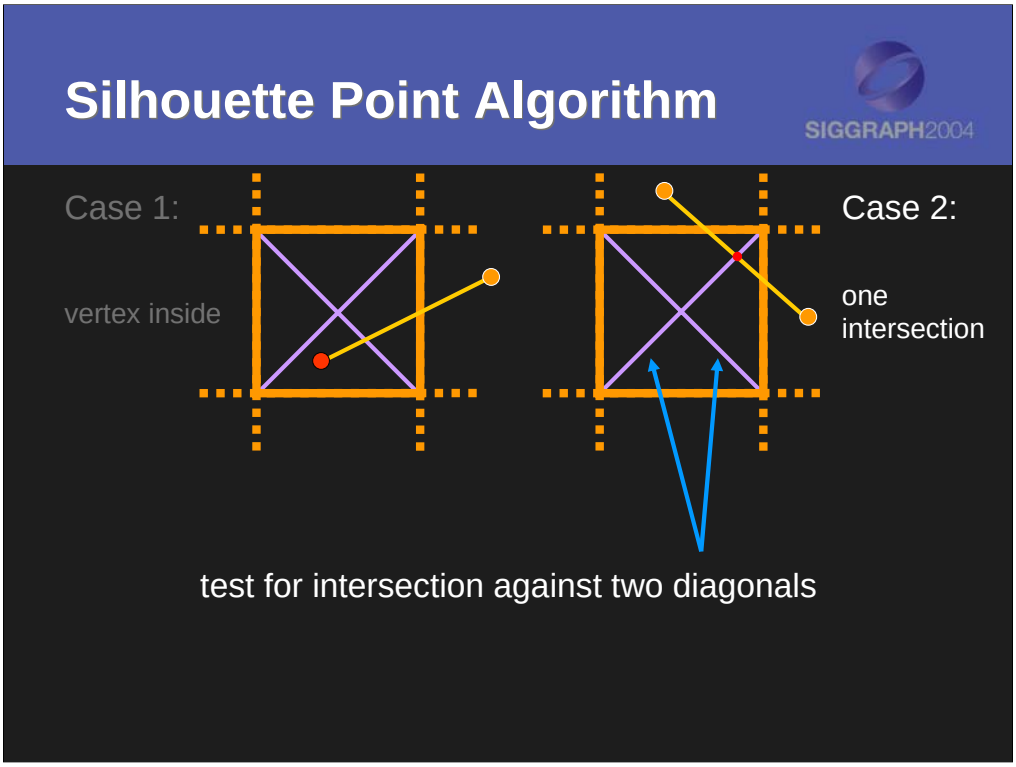
Silhouette Point Algorithm



In the first case, one of the endpoints of the edge (i.e. a vertex) lies within the fragment, as shown here. The thick orange lines represent the fragment boundary. Ignore the diagonal purple lines for the moment.

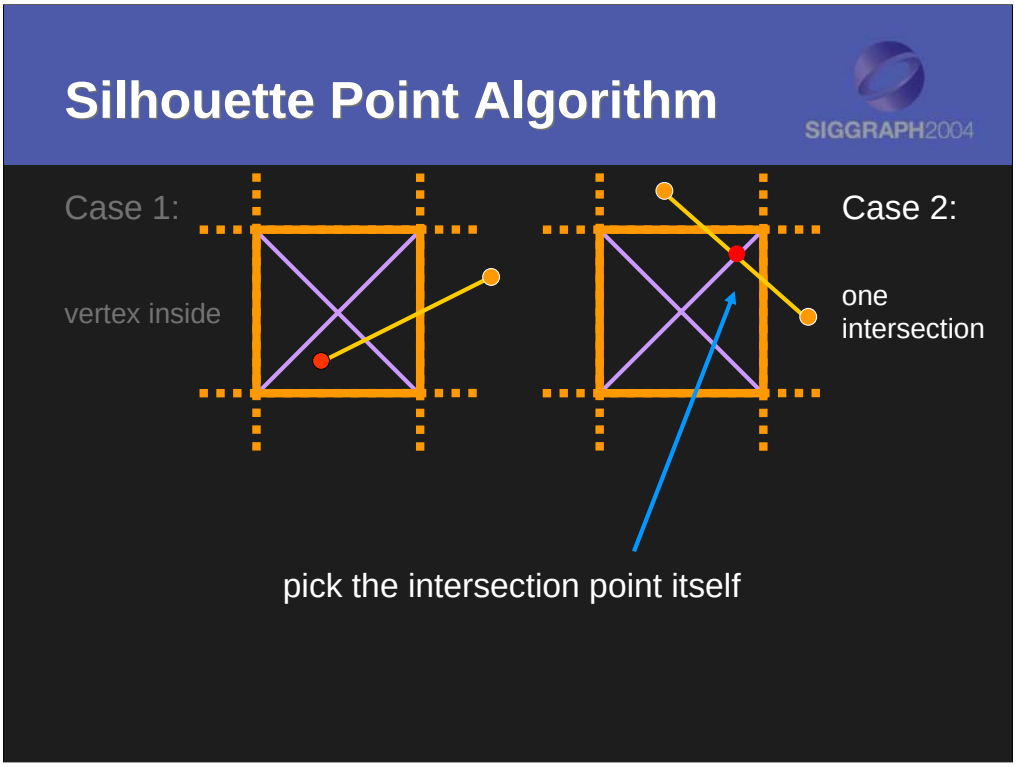


In this case, we simply pick the vertex itself as the silhouette point and store its coordinates into the silhouette map.

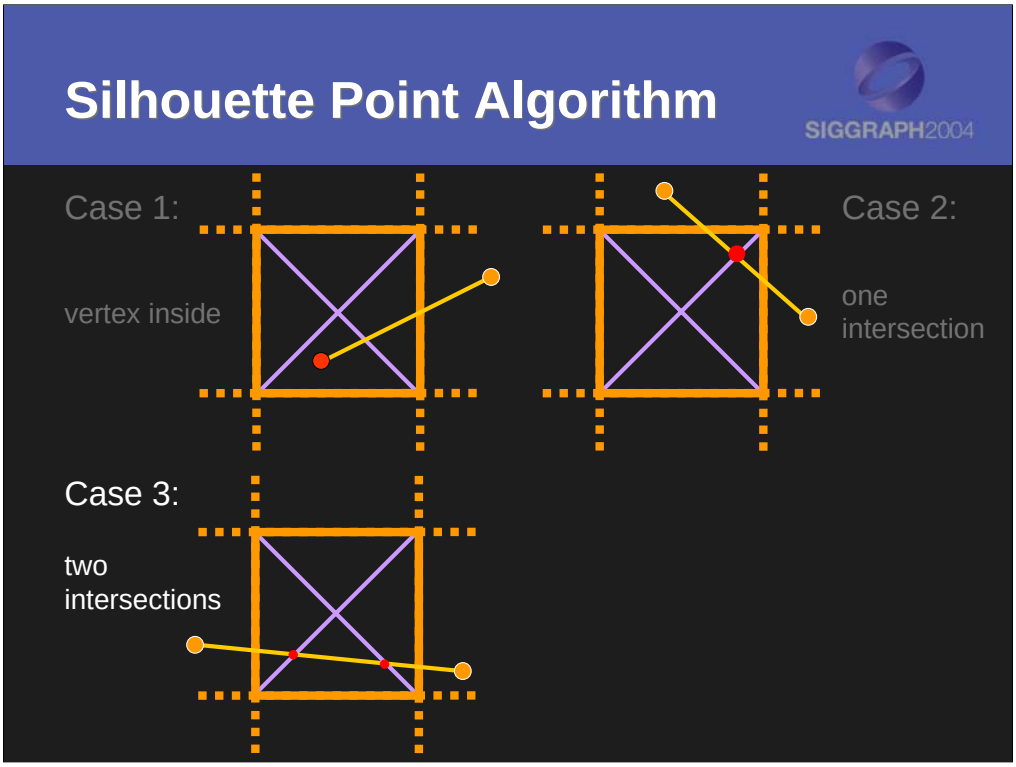


If neither endpoint of the silhouette edge lies within the fragment, we need to check for intersections. One way to do this is to perform a line segment intersection test against the two diagonals, shown in purple. Clearly, the silhouette edge will intersect the fragment if and only if it intersects at least one of the diagonals.

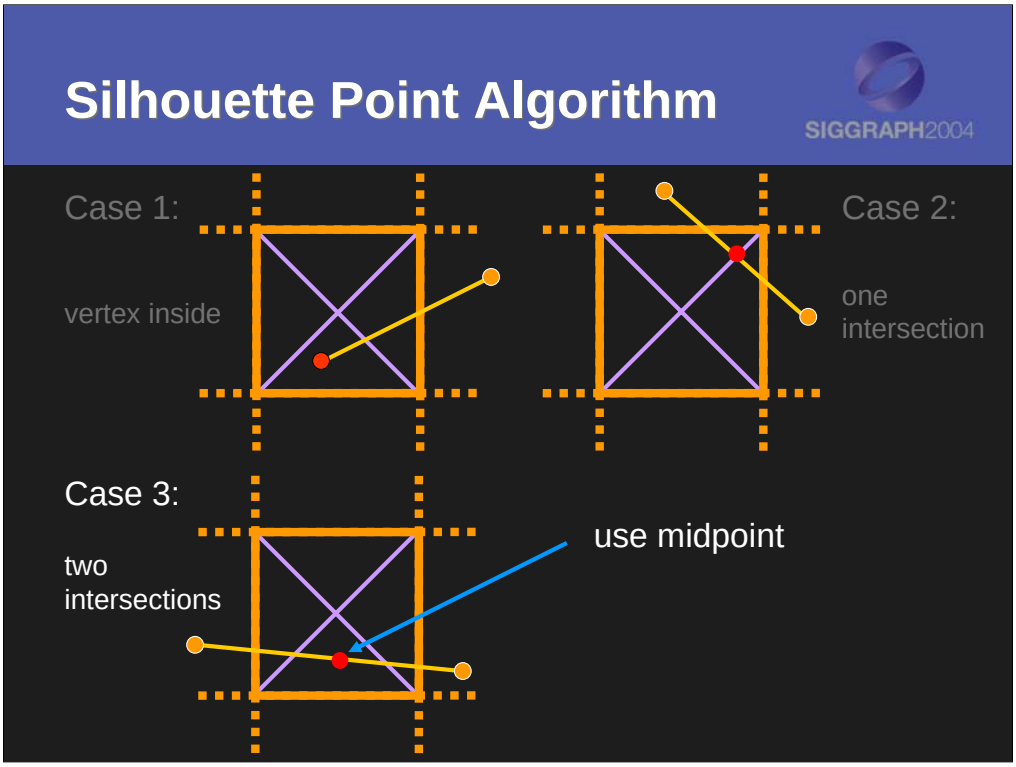
In the second case, suppose the silhouette edge intersects only one of the diagonals.



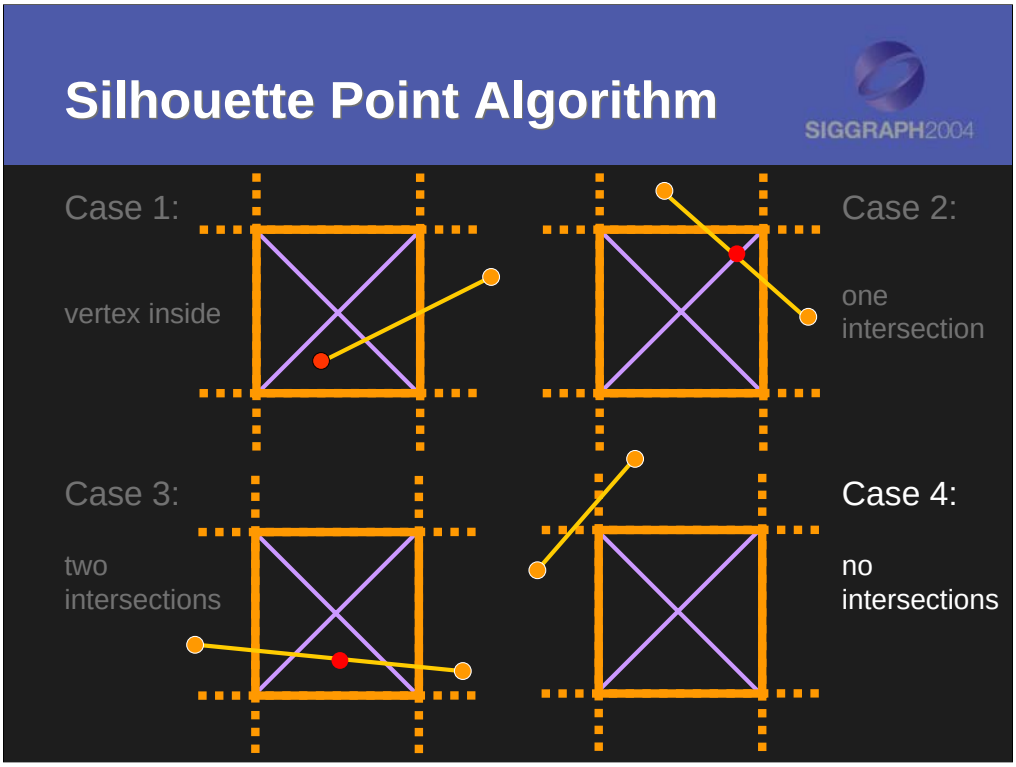
For this case, we pick the intersection point itself as the silhouette point to be stored.



In the third case, there are two intersections, one with each diagonal.

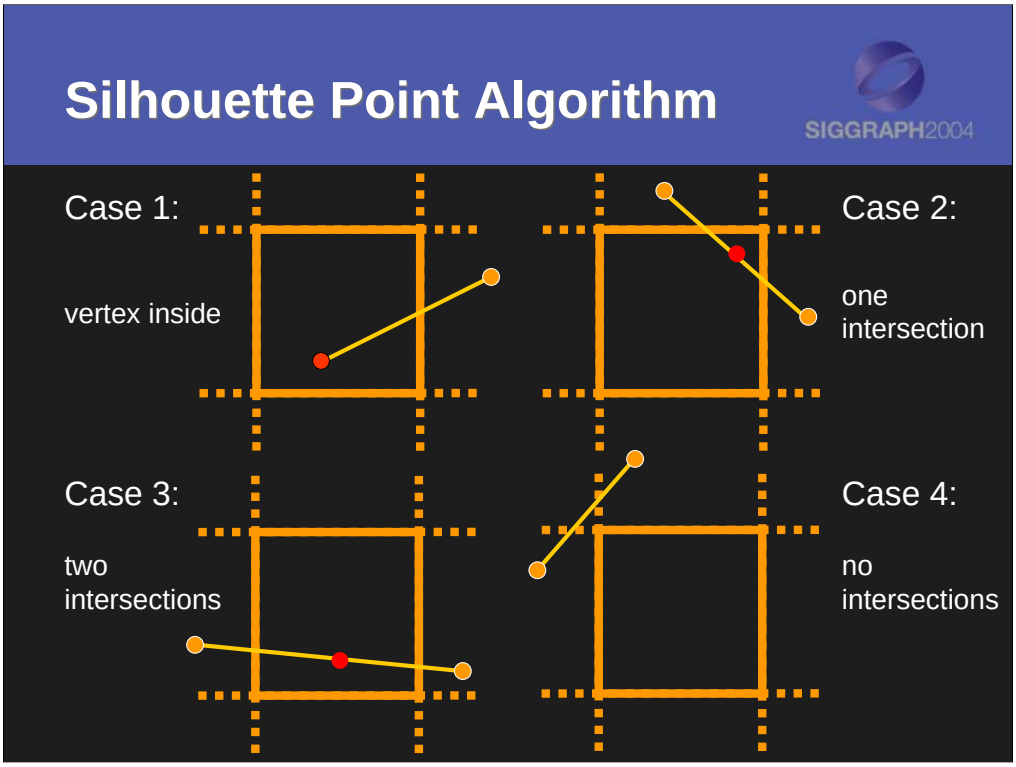


In this situation, we pick the midpoint of the two intersections as the silhouette point.



Finally, it is possible that there is no intersection between the silhouette edge and the fragment. This is possible because we are rasterizing the silhouette edge conservatively, and thus such fragments may be generated by the rasterizer.

In this case, nothing is written to the silhouette map for this fragment.



That's all there is to the silhouette-point-picking algorithm. In summary, we rasterize the silhouette edges (conservatively) and for each fragment generated, we perform the four tests shown here.

Render scene

How to compute shadows?

Split problem into two parts:

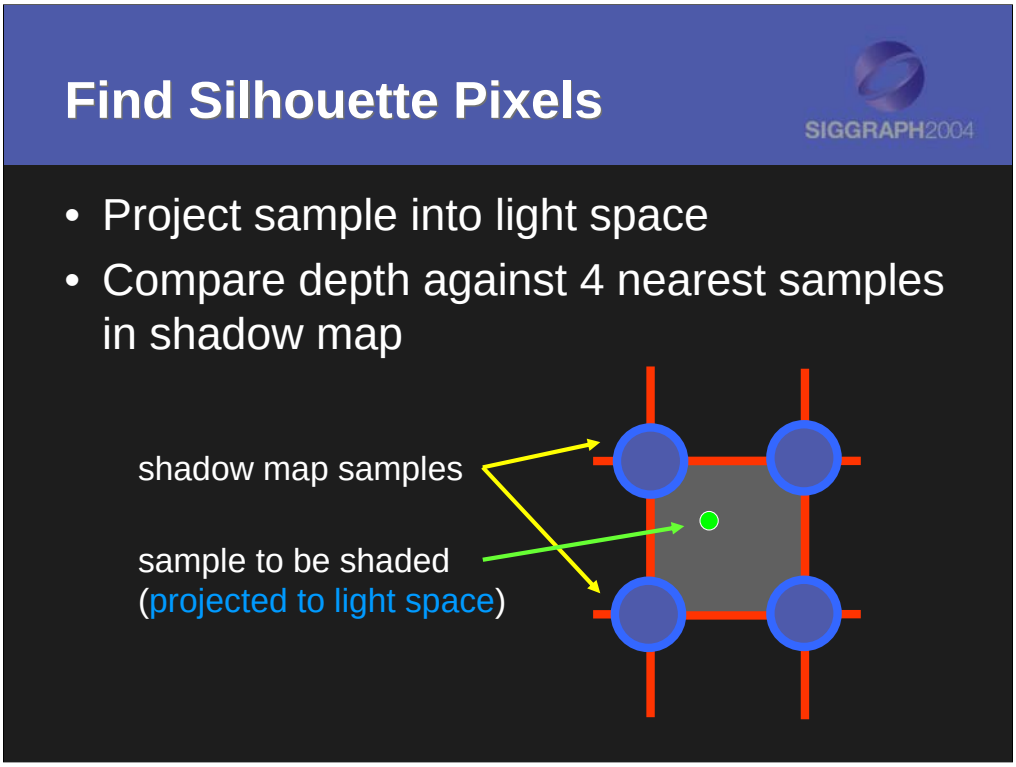
- non-silhouette pixels: use shadow map
- silhouette pixels: use silhouette map

In the final step (third rendering pass) of the algorithm, we draw the scene from the observer's viewpoint and compute shadows. How do we use the information gathered so far (a depth map and a silhouette map) to compute these shadows?

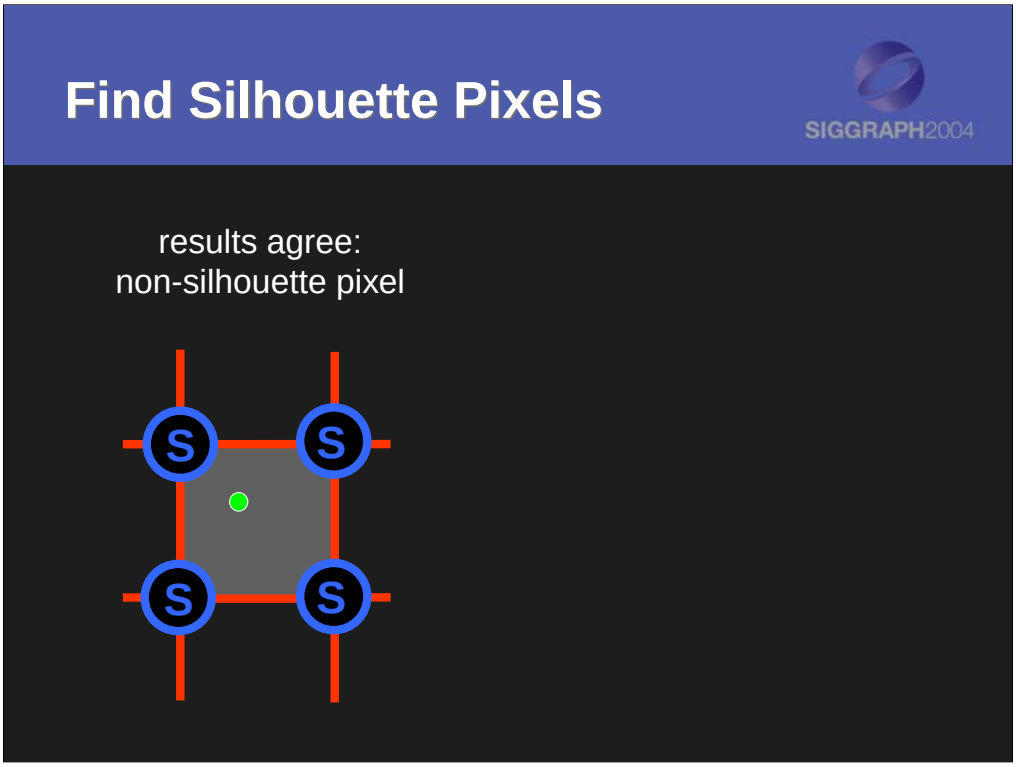
A conceptually simple way to think about the problem involves splitting the problem into two parts. Let's use the term "silhouette pixels" to refer to the pixels in the final image (seen from the observer's view) that contain shadow discontinuities, i.e. the boundary between shadowed and illuminated regions. Recall that with standard shadow maps, these pixels give us the most grief because they are precisely the ones that exhibit aliasing artifacts. All other pixels in the scene look fine because they are completely illuminated or completely in shadow.

Therefore, let's consider silhouette pixels and non-silhouette pixels separately. Since non-silhouette pixels don't show aliasing artifacts, we'll use the standard shadow map to compute shadows for those pixels. On the other hand, for the silhouette pixels, we'll use the silhouette map to obtain a good, piecewise-linear reconstruction of the shadow silhouette. We'll see exactly how to do this in a moment.

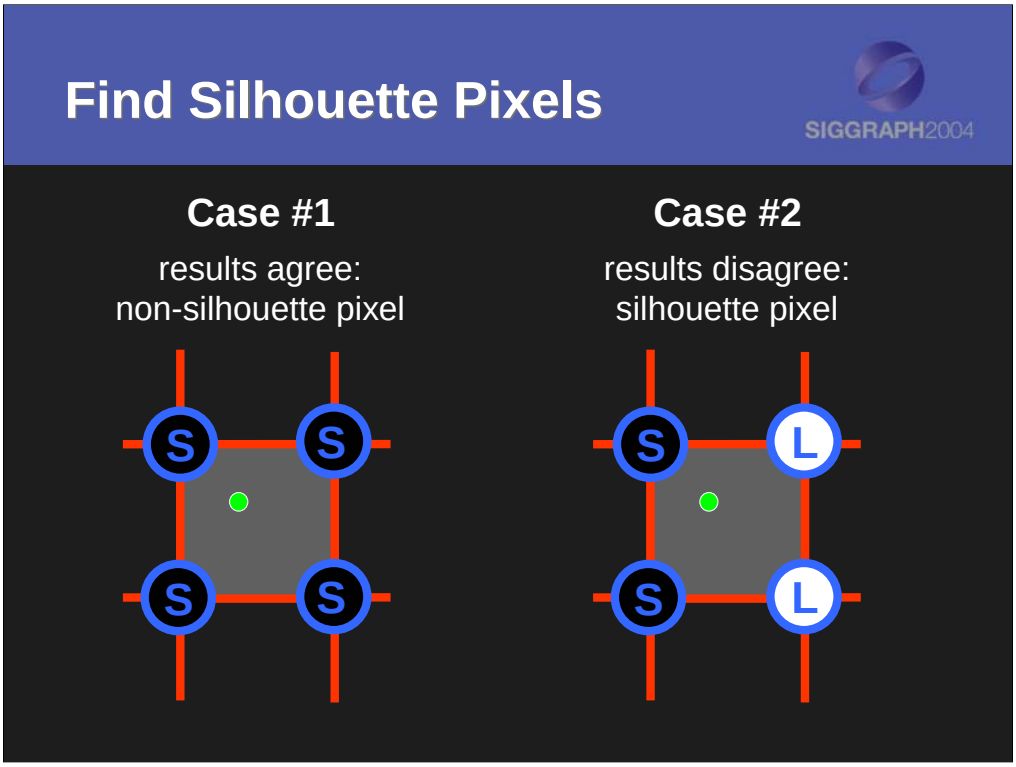
Find Silhouette Pixels



First, we need a way to identify silhouette pixels. This task turns out to be surprisingly easy. Let's say we have a sample (pixel) in the final image, and we want to know whether it's a silhouette or non-silhouette pixel. We transform the pixel into light space, just as we would do when using the standard shadow map algorithm. In general, the transformed sample (shown in green in the diagram) will lie between four samples of the shadow map (shown in blue). Compare the depth of the sample against the depths associated with the four adjacent samples of the shadow map.

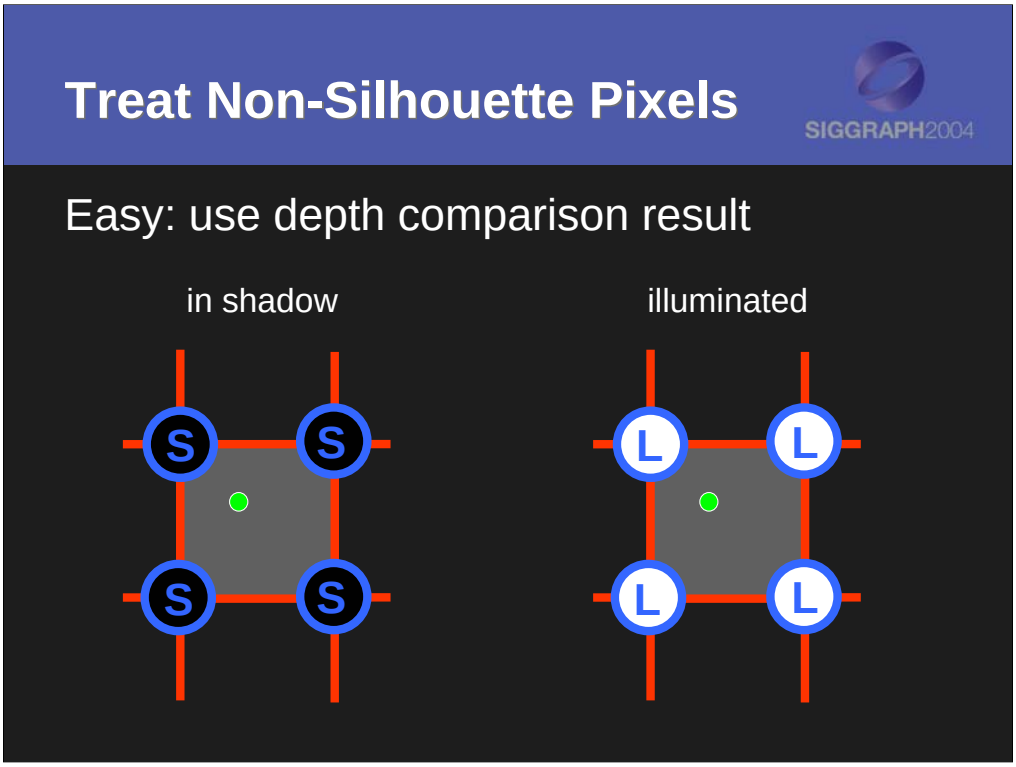


If all the depth comparison results agree, then the sample is a non-silhouette pixel. In the example shown here, all the depth comparison results indicate that the sample is in shadow (S).



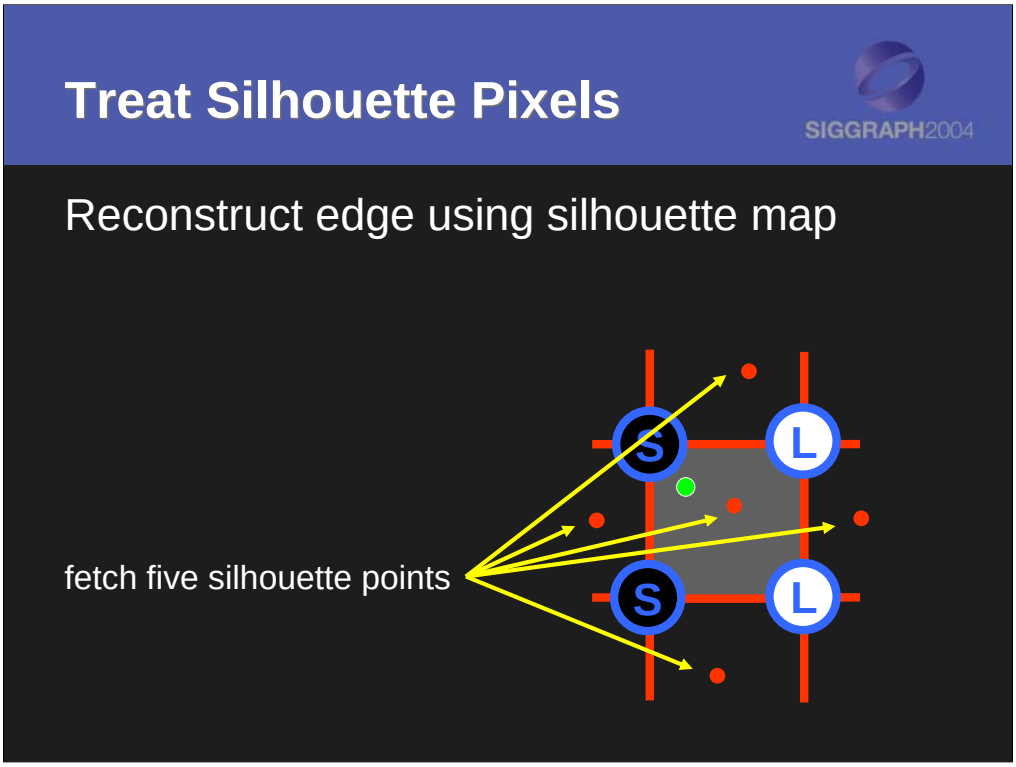
On the other hand, if the depth comparison results disagree, then the sample is a silhouette pixel. In the example here, two of the four tests declare that the sample is in shadow, but the other two declare that the sample is illuminated.

(You may have noticed this technique is very similar to the idea of Percentage Closer Filtering [Reeves et al. 1987].)



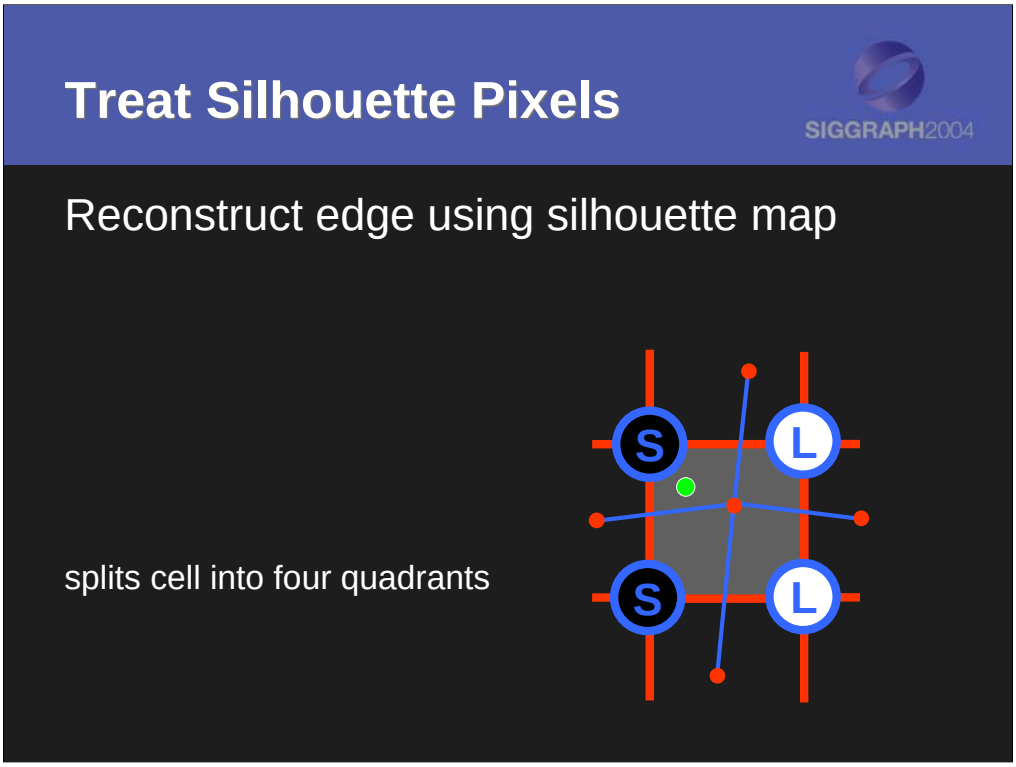
For non-silhouette pixels, computing shadows is easy. We just use the results of the depth comparisons to shade the sample. If all results say the sample is in shadow (left image) then the sample is in shadow. Similarly, if all results say the sample is illuminated (right image), then the sample should be illuminated.

Treat Silhouette Pixels



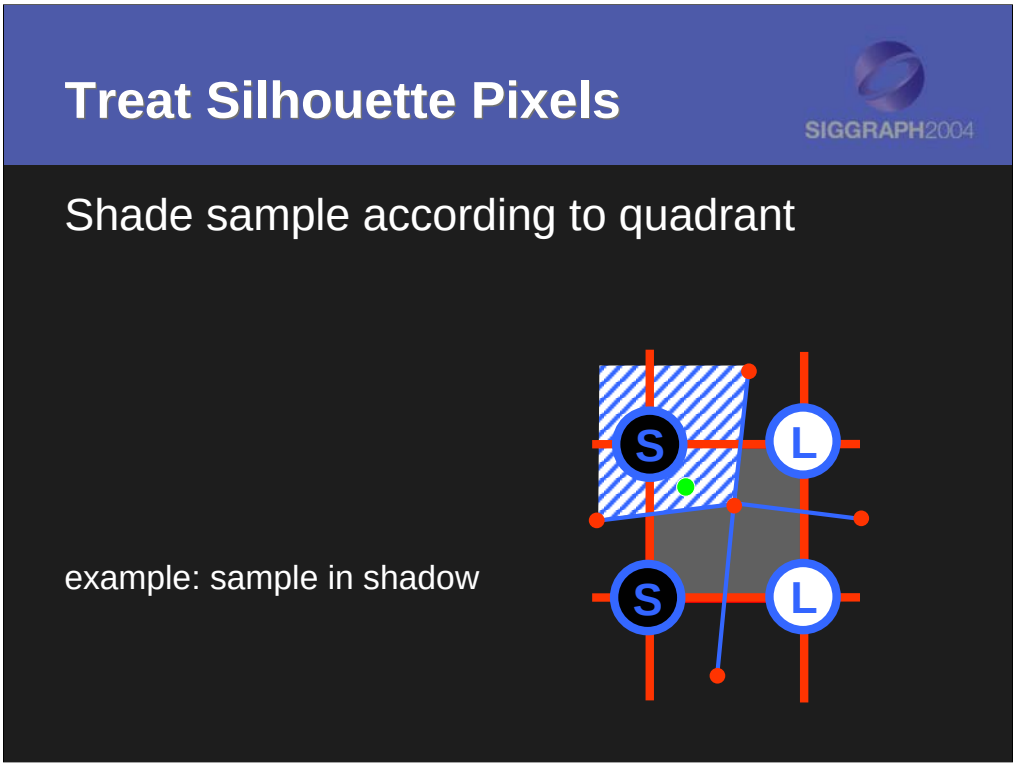
The more interesting part is handling silhouette pixels. In this case, we rely on the silhouette map to help us reconstruct shadow edges. First, transform the sample to light space (just as in the previous step) and lookup the current silhouette point and the four neighbors (shown as red points in the diagram). This essentially amounts to five texture lookups.

Treat Silhouette Pixels



If we imagine drawing line segments between these silhouette points as shown, we see that the segments split up the texel into four quadrants.

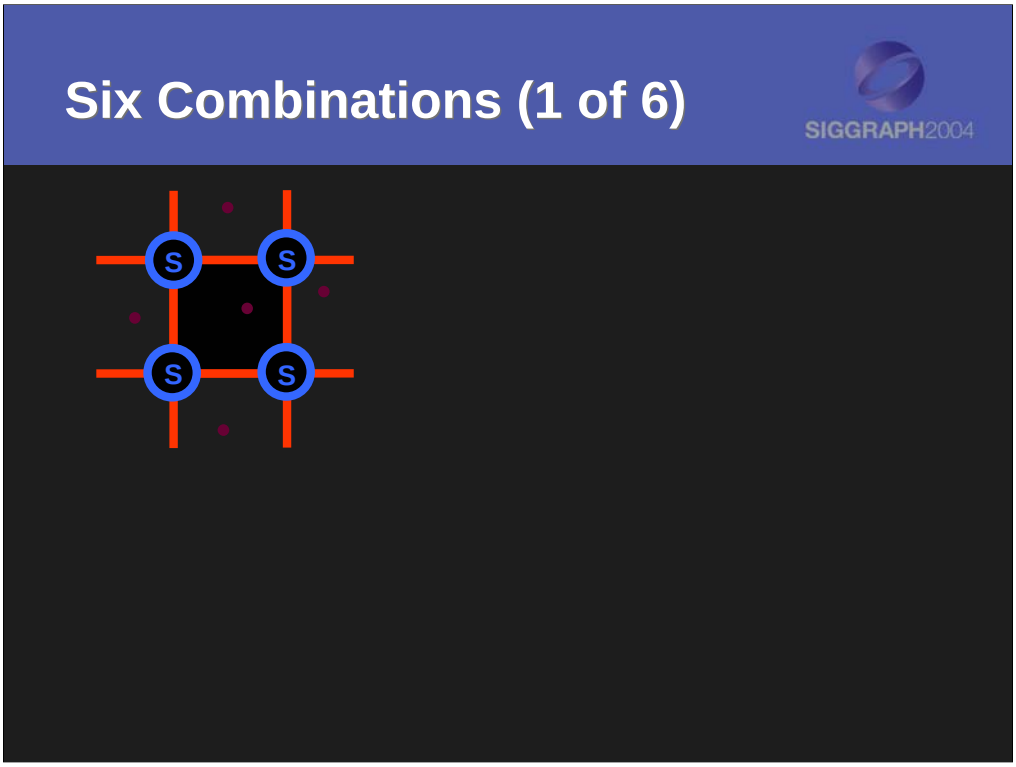
Treat Silhouette Pixels



Find the quadrant that contains the sample and shade the sample according to the depth comparison result of the associated depth sample. In the above example, the top-left quadrant contains the sample, and the depth sample associated with that quadrant indicates the sample should be in shadow. Thus we render this sample as being in shadow. If the sample had instead fallen into the top-right quadrant, it would have been illuminated.

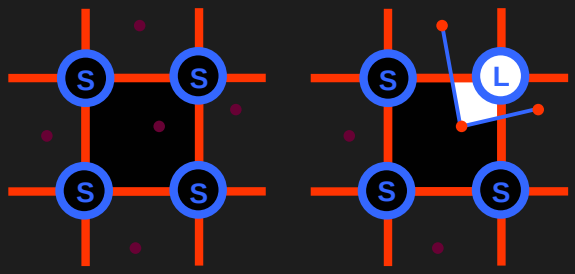
Hopefully it is clear from the diagram why we obtain a piecewise-linear approximation to the shadow edge, as opposed to the previous piecewise-constant approximation (using a regular shadow map). The quadrants essentially define different shadow boundaries within a single texel, i.e. at sub-texel precision. All samples that fall into a given quadrant will be shaded the same. In contrast, with a regular shadow map, all samples that lay within the texel itself are shaded the same.

Six Combinations (1 of 6)

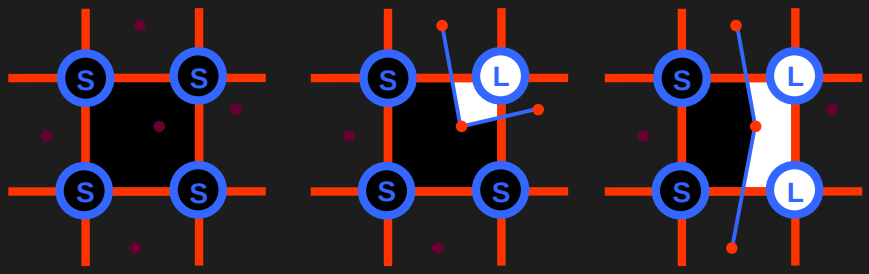


For clarity, let's consider all the possible cases that can arise. There are only six of them. The first possibility, shown here, is when all four depth samples indicate the current sample (i.e. the sample to be shaded) is in shadow. (This actually falls into the case of the non-silhouette pixels, which we covered earlier.)

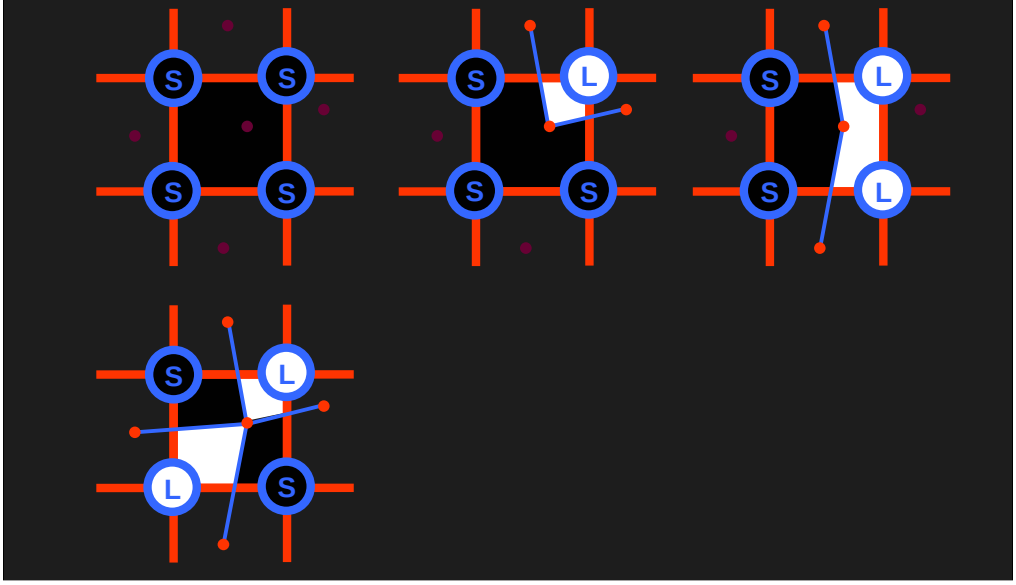
Six Combinations (2 of 6)



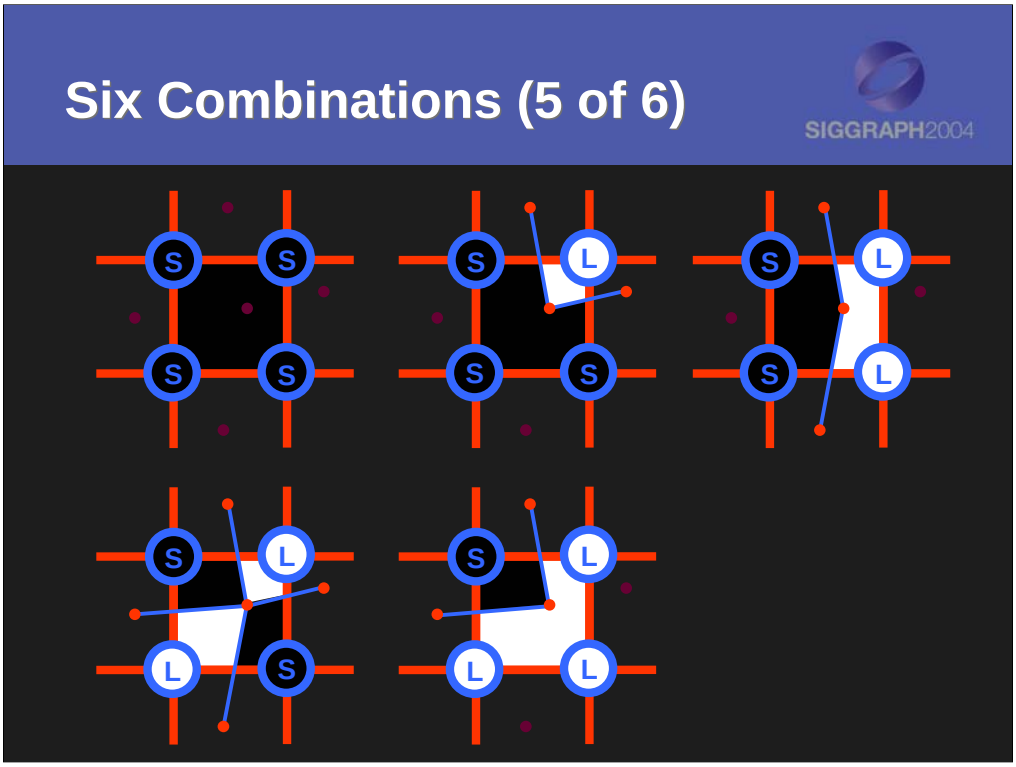
Six Combinations (3 of 6)

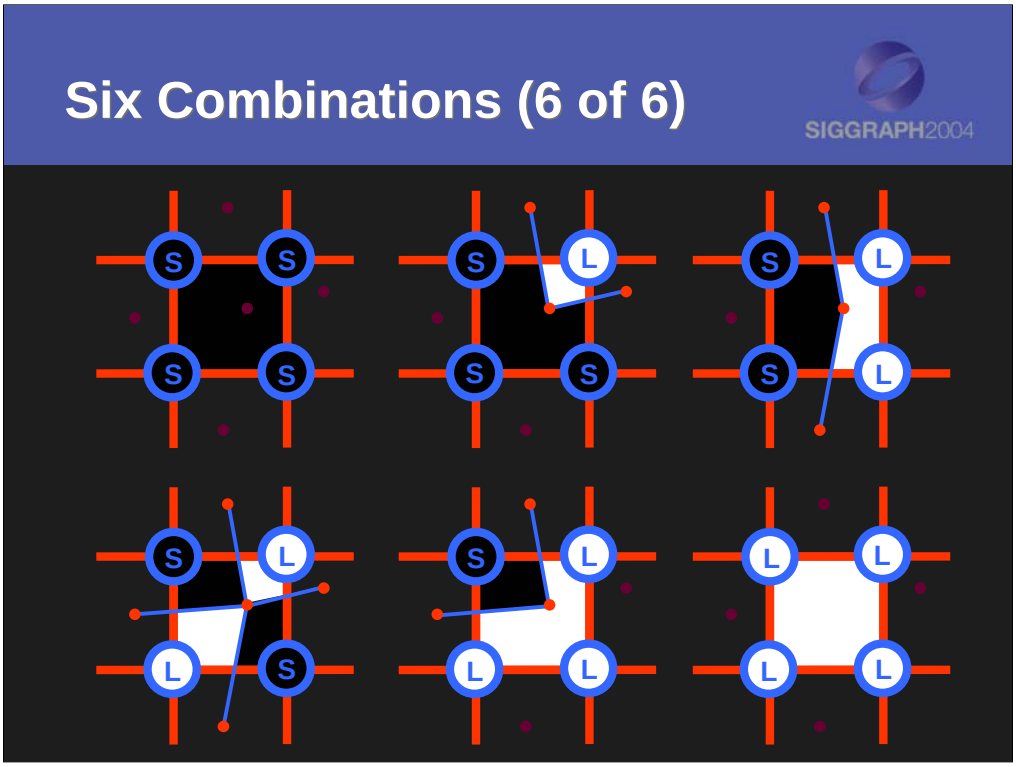


Six Combinations (4 of 6)



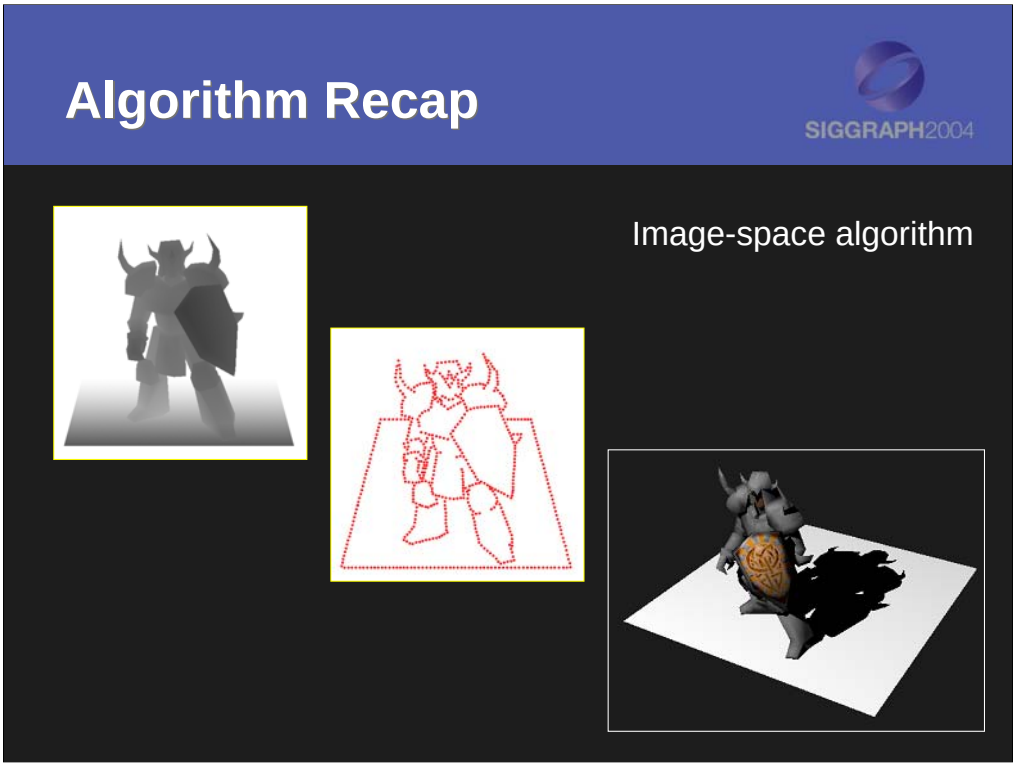
Six Combinations (5 of 6)





In the final case, all depth comparisons agree and the sample is illuminated. Again this falls into the case of non-silhouette pixels.

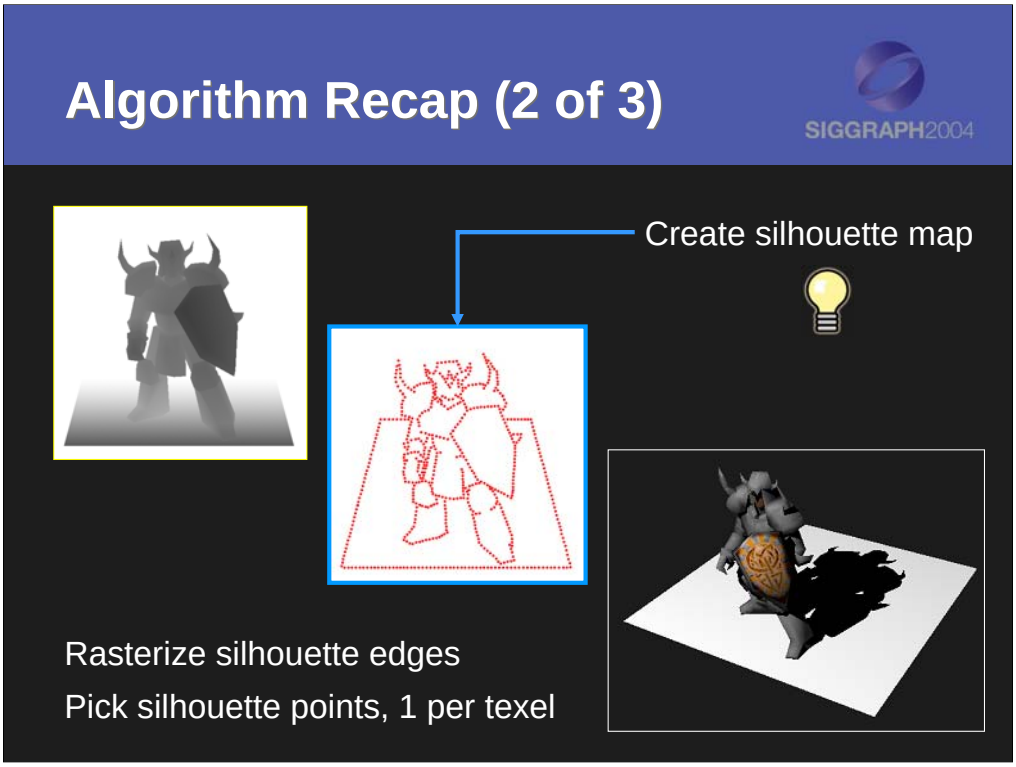
In summary, there are 2 cases (top-left, bottom-right) for non-silhouette pixels and 4 cases for silhouette pixels.



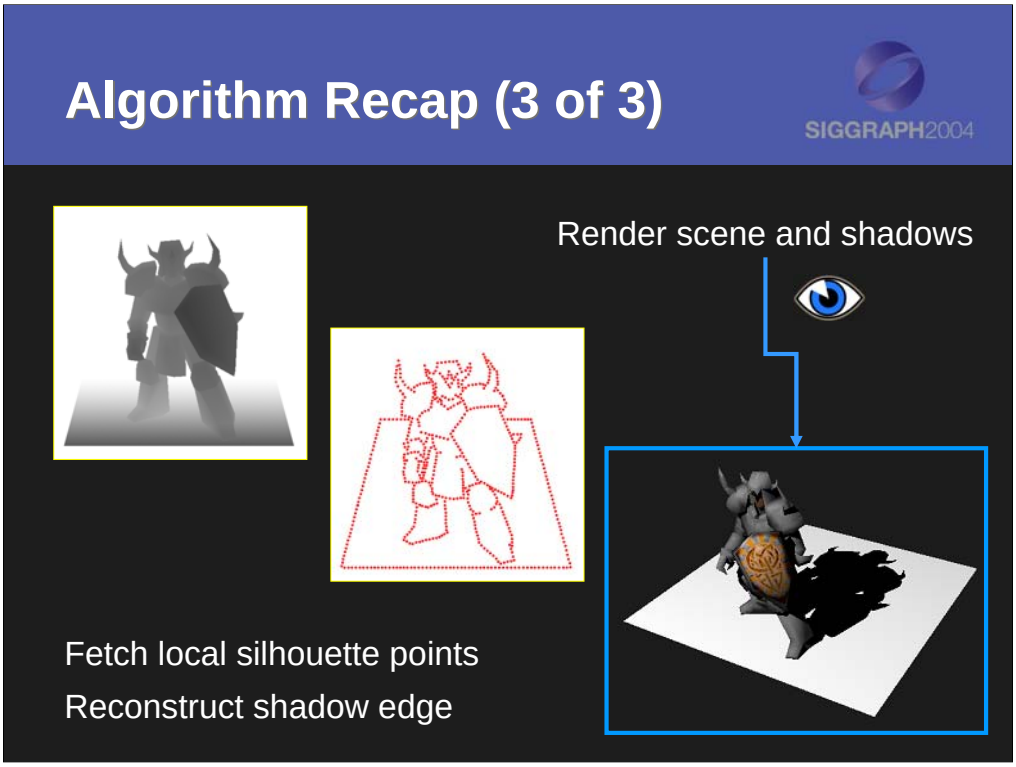
We have now covered all the algorithm's details. Let's take a step back and review the algorithm at a higher-level.

First, note that the algorithm works in image space. All relevant information (depth samples, silhouette points) are stored using 2D image representations. The silhouette map is really just an image-based edge representation.

The first step is to create a depth map from the light's viewpoint.



The second step is to create the silhouette map, also from the light's viewpoint. During this step, we rasterize silhouette edges conservatively and pick points that lie exactly on the edge and store the coordinates of these points into the silhouette map. The idea is to check for edge-fragment intersections by checking against the two diagonals of the fragment. There are a few simple cases to consider, as described in detail earlier.



In the final pass, we draw the scene from the observer's viewpoint and draw the shadows. For non-silhouette pixels, just use the shadow map. For silhouette pixels, we fetch five silhouette points (current point + four neighbors) and use these points to reconstruct a piecewise-linear approximation to the true shadow silhouette.

Up till now, we've been discussing the concepts of the algorithm. It's time to see how we can implement the algorithm on modern graphics hardware.

Implementation

- Details (OpenGL)
- Hardware acceleration
- Optimizations

The silhouette map algorithm can be implemented on DirectX 9-class hardware. This means specifically that you need to have programmable vertex and fragment units, and floating-point precision (at least 16 bits of floating-point) must be available in the programmable fragment unit. This precision is necessary for a number of tasks we have to perform. For instance, we need to perform intersection tests when generating the silhouette map. Examples of suitable hardware include the ATI R300 chips (e.g. Radeon 9700 and later) and the NVIDIA NV30 chips (e.g. GeForce FX and later).

The silhouette map algorithm can be implemented using both OpenGL and DirectX. However, any code snippets I show here will be in OpenGL.

Create Shadow Map

Render to standard OpenGL depth buffer

Optimizations

- for closed models, cull back faces
- turn off shading, color writes
- only send vertex positions
- draw roughly front-to-back

To create a shadow map, we place the OpenGL camera at the light position of the light source, aim it at the scene, and draw. Keep in mind there are a number of optimizations that we can perform. For closed models, turn on back-face culling, e.g.

```
glEnable(GL_CULL_FACE);
```

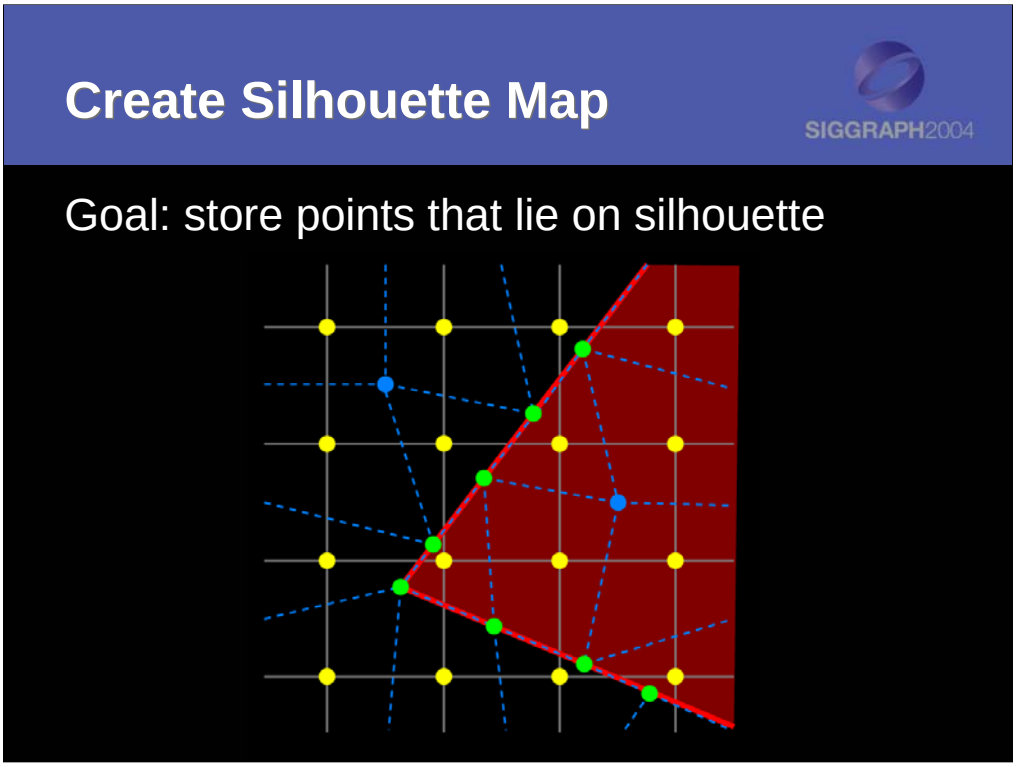
since those faces won't be seen anyways. In addition, since we only care about drawing depth values, we don't have to perform shading. Therefore, turn all fancy shaders off. Furthermore, we don't even have to write anything to the color buffer, so turn off color writes:

```
glColorMask(0, 0, 0, 0);
```

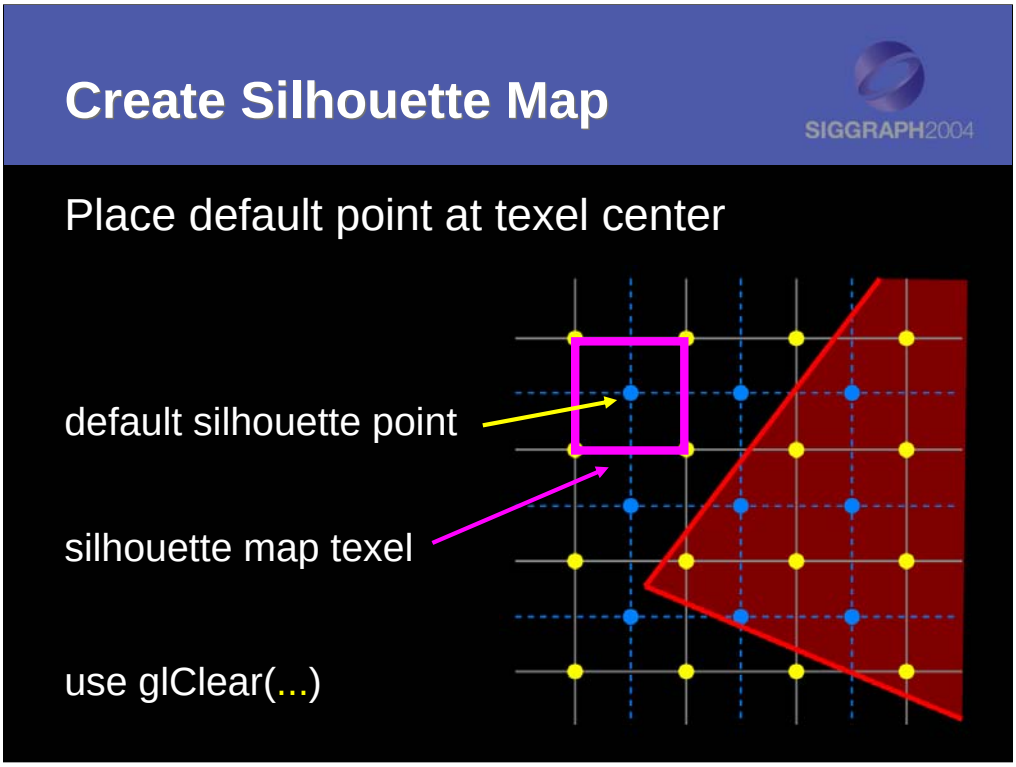
In terms of transferring data from the host processor (CPU) to the graphics processor (GPU), we only need to send the vertex positions. Since we're not doing any shading, don't send extra information like texture coordinates and normals.

Finally, draw the objects roughly in front-to-back order. Doing so maximizes the hardware's ability to perform early Z rejection (i.e. occlusion culling).

Create Silhouette Map



Now let’s see how to compute the silhouette points in the second step of the algorithm.

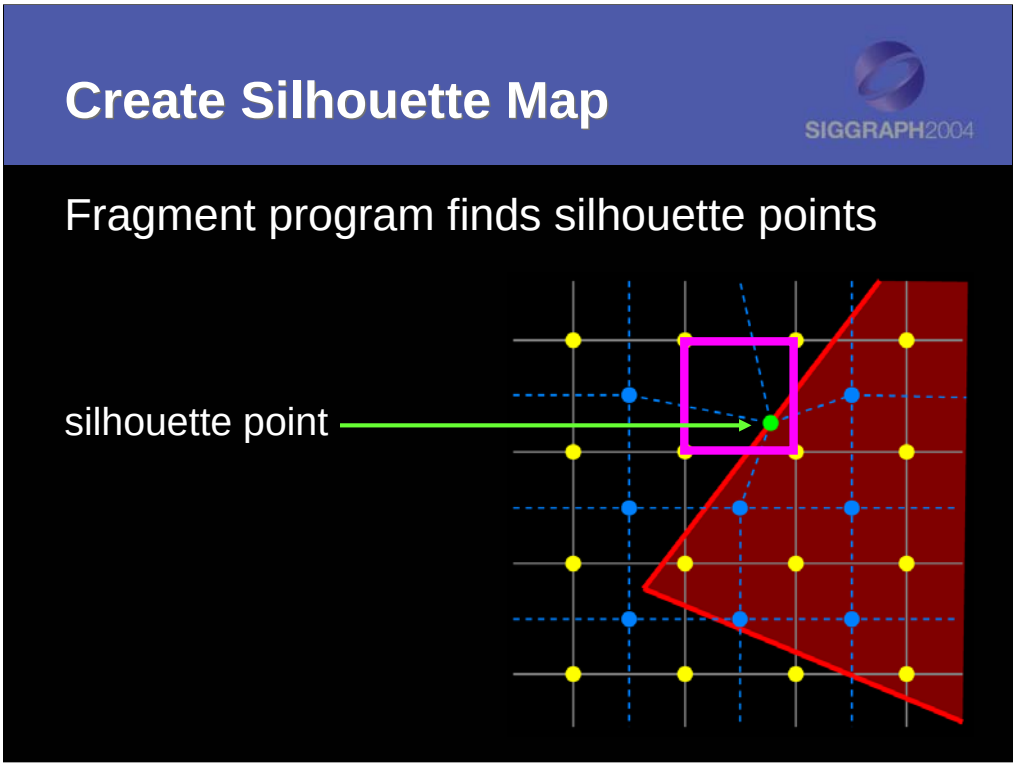


Remember that the silhouette map is the dual grid of the depth map and is offset from the depth map by ½ a pixel. Earlier, we discussed the concept of having a deformed depth mesh in which the depth samples are moved to lie along silhouettes. Our strategy for creating the silhouette map will be as follows. First, let's start with an undeformed depth mesh, meaning that all depth samples lie at their original, undeformed positions. Then we'll rasterize silhouette edges and compute silhouette points to perform the deformation on some of the depth samples.

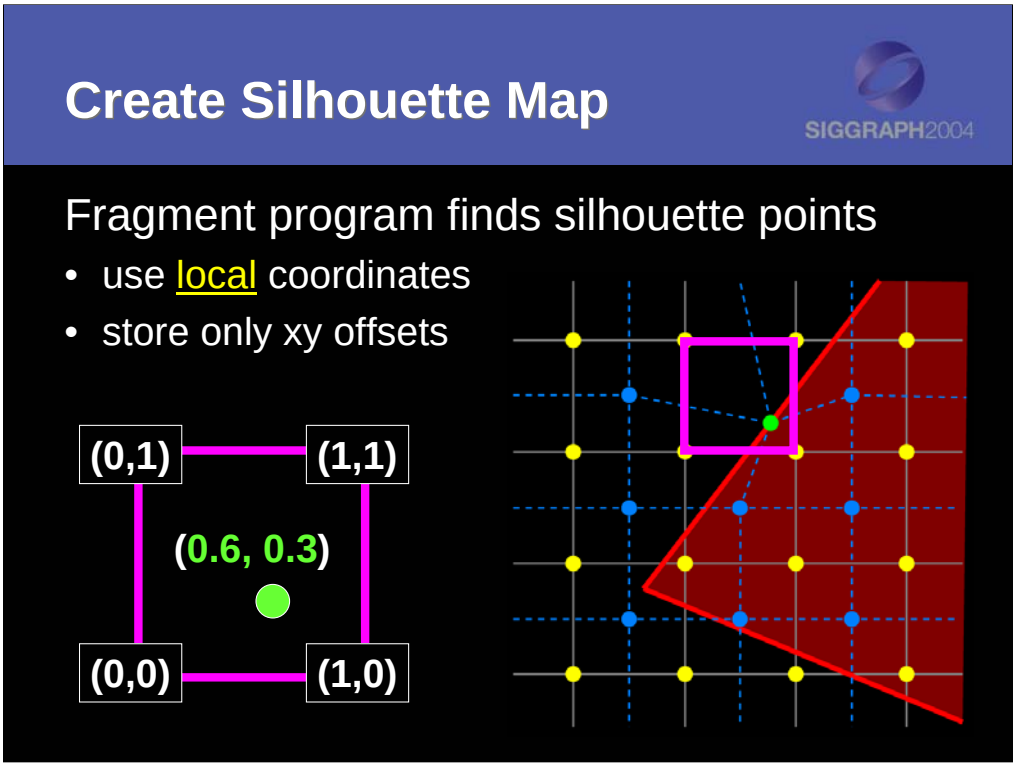
To implement this, we start by placing a default silhouette point at the center of every texel of the silhouette map. This essentially places the silhouette points directly on top of the depth samples of the depth map. This construction is shown in the diagram. The purple box represents the boundary of a single texel of the silhouette map. The blue dots are the locations of the depth samples. We initialize the silhouette map by placing silhouette points at these blue points.

In practice, it's easy to perform this initialization by using the glClear call to clear the whole buffer.

Create Silhouette Map



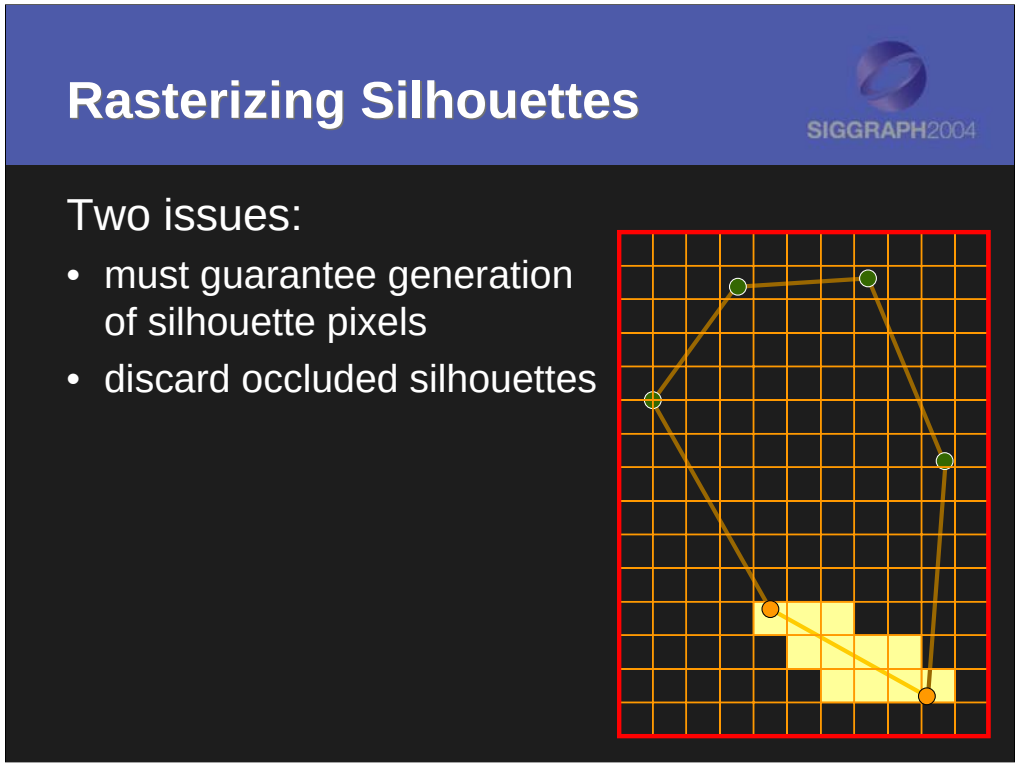
We'll use a fragment program (pixel shader) to compute the silhouette points for texels that contain silhouette edges. To make this concrete, consider the purple texel shown in the diagram. A fragment associated with this texel will be generated by the rasterizer. We want a fragment program that checks that a silhouette edge passes through this fragment, computes the silhouette point, and writes it to the output buffer (the silhouette map).



The fragment program simply performs the various intersection tests discussed earlier in the silhouette-point-picking algorithm. For fragments that are generated by the rasterizer but not crossed by a silhouette edge, the fragment program uses a “fragment kill” to throw away the fragment (i.e. write nothing to the output).

One way of computing and storing the silhouette points is to use a local coordinate system, in which the texel area is taken to be a unit square (see the lower-left diagram). The default silhouette point is at the center, (0.5, 0.5). For silhouette points computed via intersection tests, we just store xy offsets in the local coordinate system into the silhouette map. These offsets are in the range [0,1].

Rasterizing Silhouettes



There are two issues to be aware of when rasterizing silhouette edges. All along we've talked about performing conservative rasterization to guarantee that all fragments crossed by a silhouette edge will be generated. Now we'll see how to do that. The second issue is that some silhouette edges, seen from the light's viewpoint, will be hidden by blockers (occluders). Since the fragments from these edges aren't seen by the light, we don't want to process them.

Rasterizing Silhouettes

Rasterize conservatively

- Be careful using OpenGL wide lines
- Use width of at least 3

glLineWidth(3);

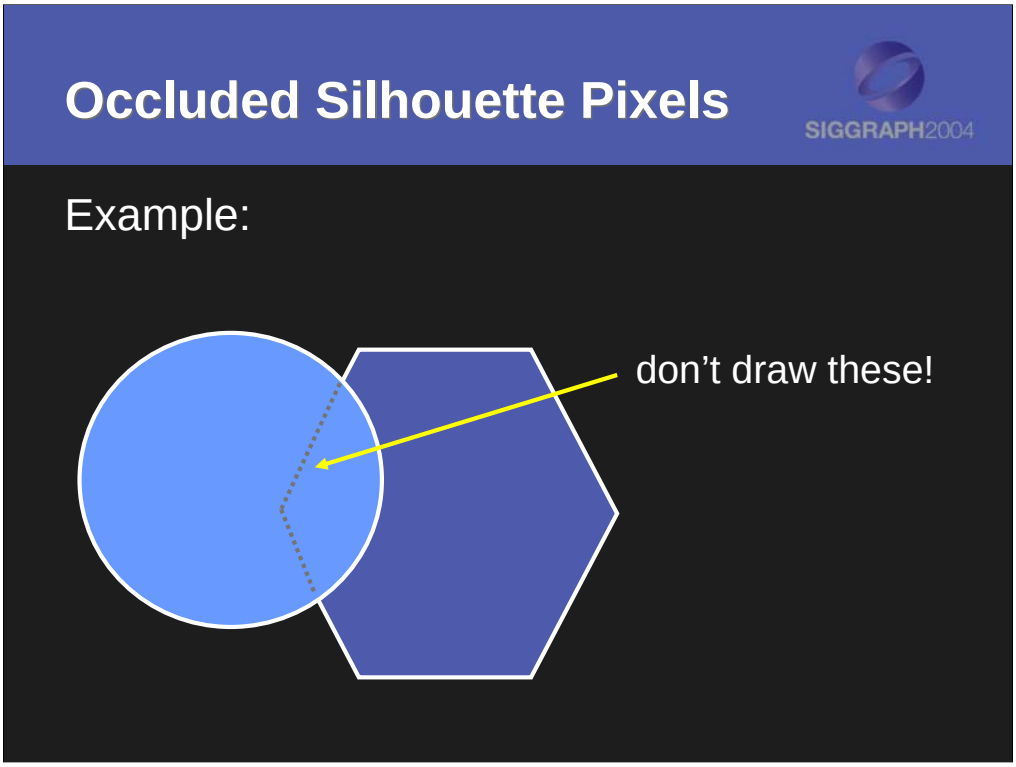
- Make lines slightly longer to cover endpoints

Another solution: use thin quads, not lines

- See Sen et al. [SIG2003] paper

Let's address the conservative rasterization first. Sen et al., in their original paper on shadow silhouette maps, recommend drawing "thin quads" – thick enough to guarantee the generation of all fragments crossed by a silhouette. Their reason for doing so is that the alternative, drawing wide lines in OpenGL, may vary in behavior across different graphics hardware.

In turns out, however, that by choosing a line width that is large enough, all necessary fragments will in fact be generated. In practice, I've found that using a width of at least 3 works consistently. Another detail to remember is that you have to make the line slightly longer than the original edge to guarantee that the fragments containing the endpoints of the edge will also be generated.

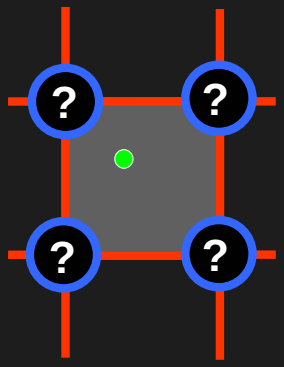


The second issue is dealing with occluded silhouette pixels. In the example shown here, the hexagon is partly occluded by the circle, seen from the point of view of the light source. The fragments belonging to the occluded silhouette edges, shown as dotted gray lines, should be ignored.

Occluded Silhouette Pixels

Implementing occlusion:

- Use depth map from first pass
- Recall silhouette map offset by ½ pixel
- Use fragment kill if depth is greater than 4 nearest samples in depth map



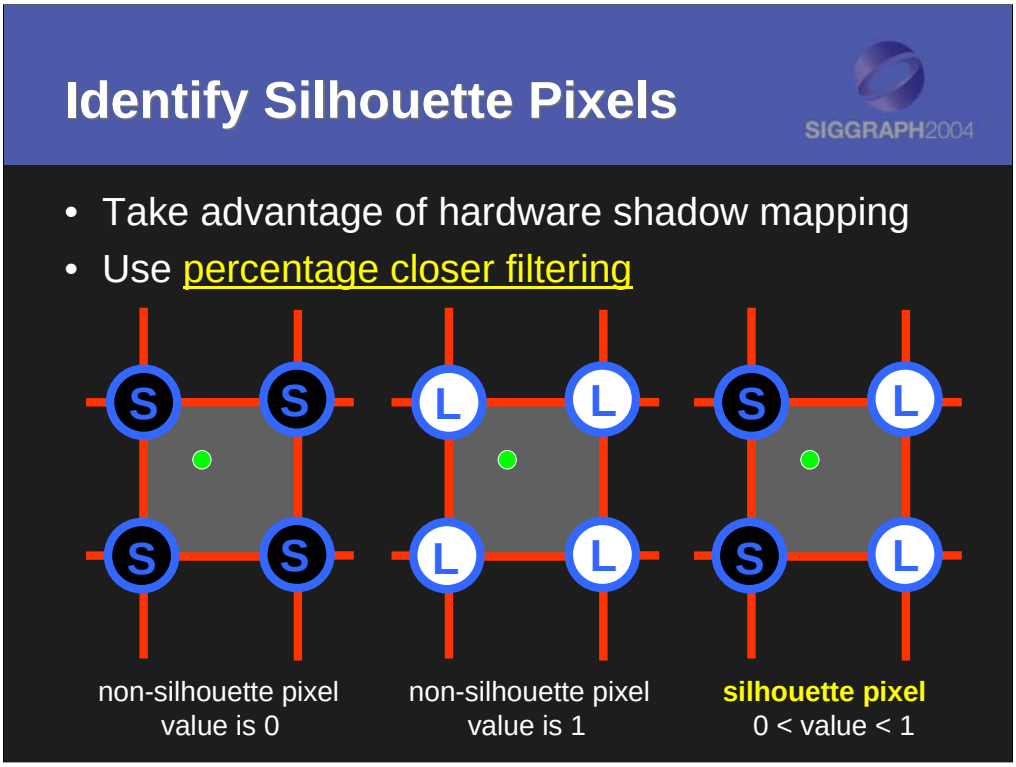
This case is easy to check for, because we already have a depth map of the blockers from the first rendering pass. In the fragment program that computes silhouette points, we also perform the following check. If the depth of a fragment belonging to a silhouette edge lies behind all four of the neighboring depth samples in the shadow map, then this fragment is occluded and should be discarded. To throw away the pixel, issue a fragment kill.

Rendering Final Image

Recall

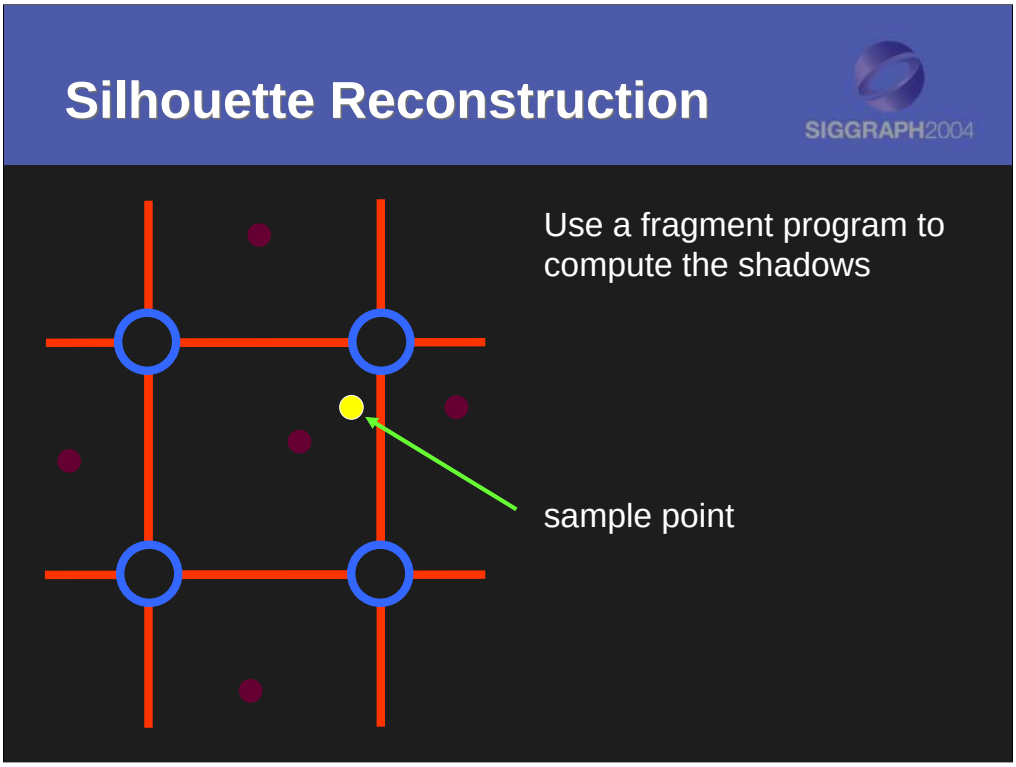
- Draw from observer's view
- Identify silhouette vs. non-silhouette pixels
- Use shadow map for non-silhouette pixels
- Use silhouette map for silhouette pixels

In the final rendering pass, we need to distinguish between silhouette and non-silhouette pixels. Just as a reminder, earlier we saw how to accomplish this by transforming a sample into light space and checking its depth against the 4 nearest samples of the shadow map. If the depth comparison results agree, then the pixel is a non-silhouette pixel. Otherwise, it's a silhouette pixel.

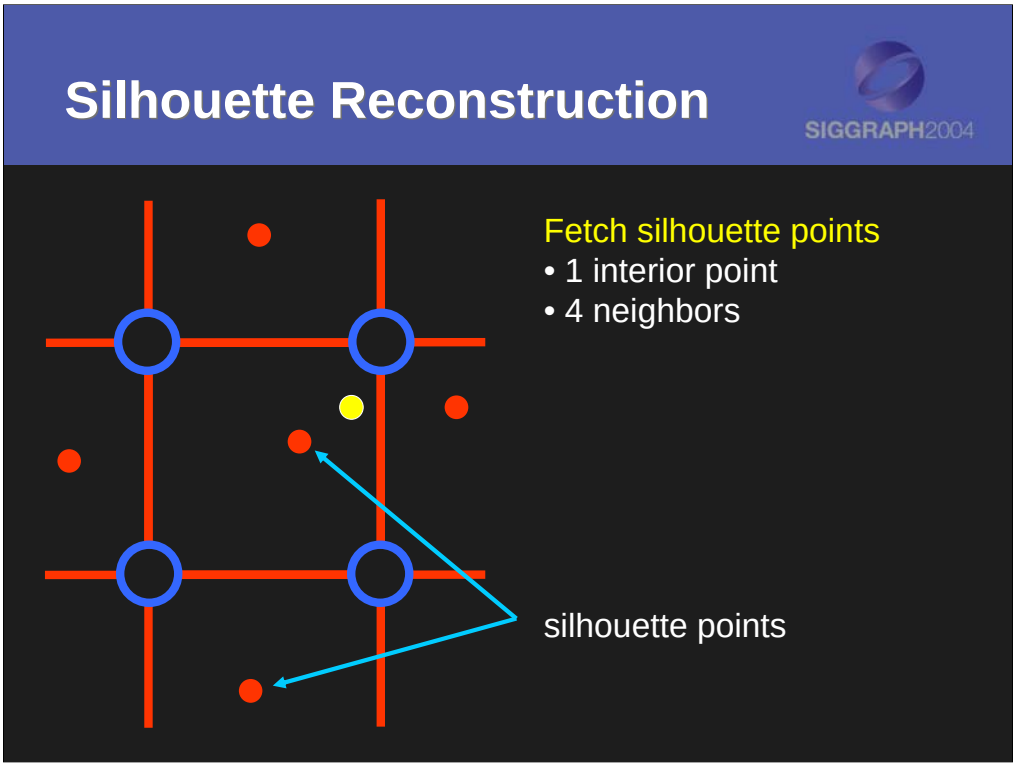


Implementing this step is a breeze using graphics hardware. The hardware supports percentage closer filtering (see Reeves et al. [1987]), meaning that instead of performing a single depth comparison of a sample against the depth map, it actually compares the depth against the 4 nearest depth samples and filters the binary results. This means that if the depth comparison results agree, then the final result will be either 0 (for a shadowed pixel) or 1 (for an illuminated one). In contrast, if the depth comparison results disagree, then, since the results are filtered, the final value will lie in between 0 and 1.

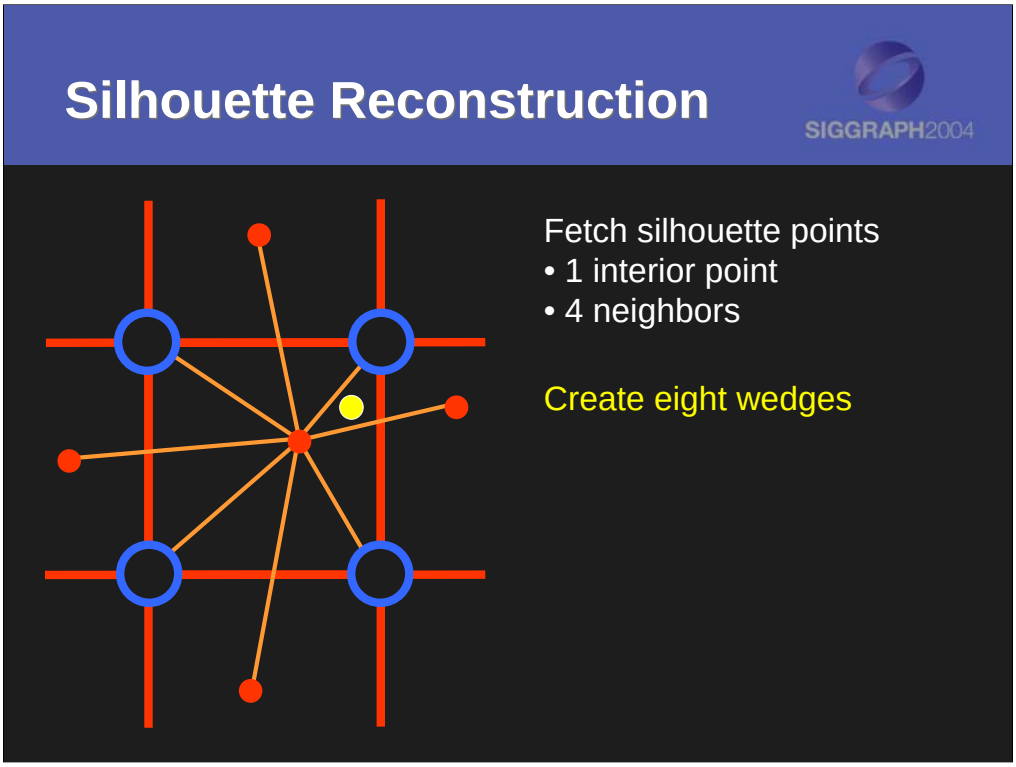
The nice thing is that this entire operation can be performed using a single shadow map (texture) lookup in a fragment program. The hardware takes care of performing the depth comparisons and returns the filtered result to your fragment program. This is both simple and fast.



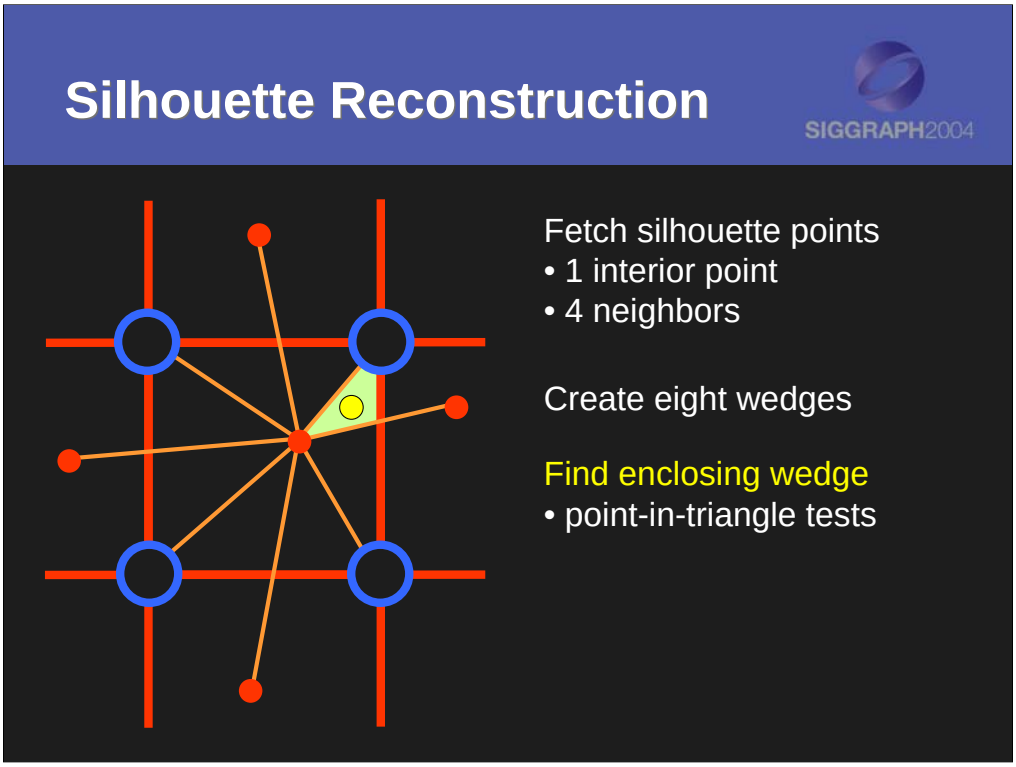
For silhouette pixels, we also use the fragment program to perform accurate shadow edge reconstruction. Let's see exactly how it works.



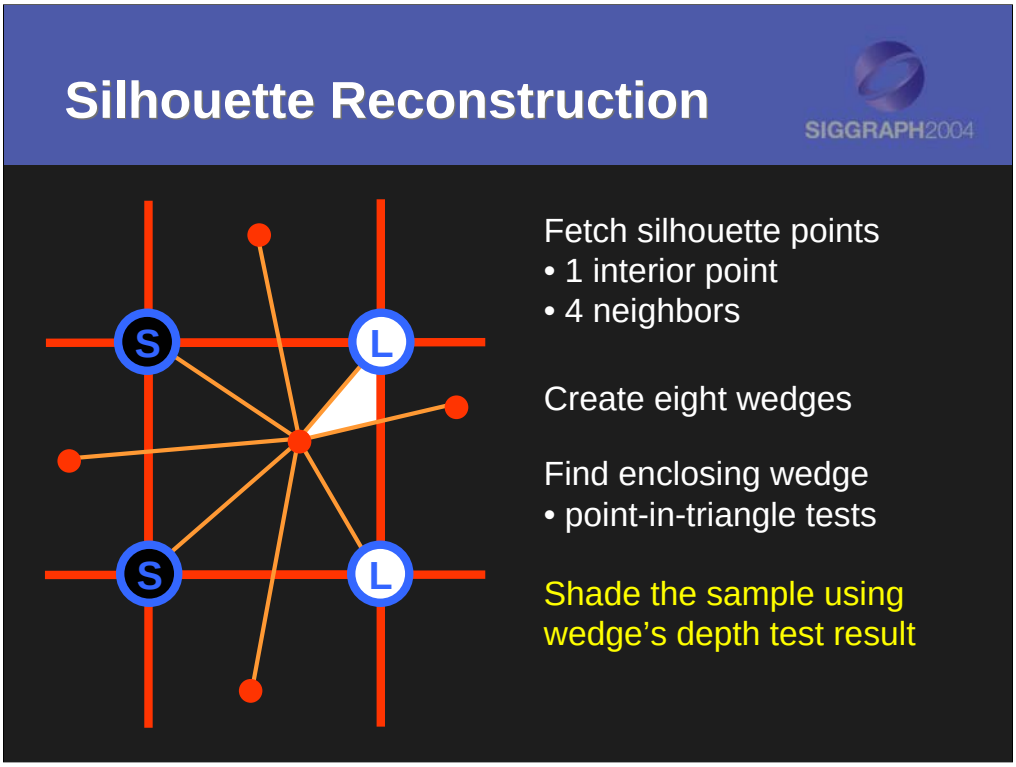
First, project the sample into light space, which maps the sample to a particular texel in the silhouette map. We fetch the silhouette points from the silhouette map, 1 point for the current texel, and its four immediate neighbors. This amounts to five texture fetches.



Now, recall that we want to know which of the four neighboring depth samples should be used for the depth comparison. Imagine using the five silhouette points and the four depth samples to carve up the texel into eight wedges (shown in orange in the diagram).

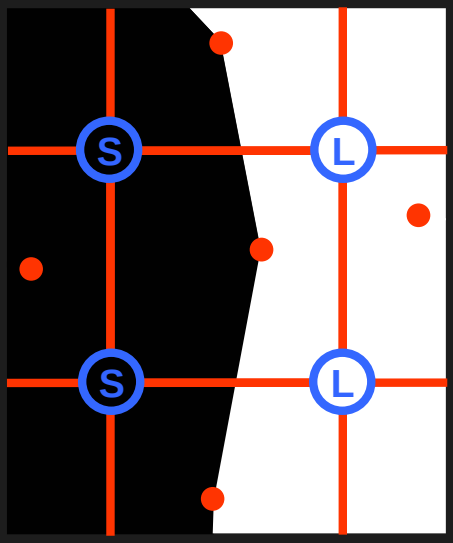


Then we simply find which wedge contains our sample. This amount to performing point-in-triangle tests.



We shade the sample according to the depth comparison result associated with the wedge. In the example here, the relevant wedge is associated with the top-right depth sample, and the depth comparison against that sample indicates that the sample should be illuminated.

Silhouette Reconstruction



Fetch silhouette points
- 1 interior point
- 4 neighbors

Create eight wedges

Find enclosing wedge
- point-in-triangle tests

Shade the sample using wedge's depth test result

Repeat for all samples

Repeat this step for all samples in the image. That's all there is to computing the shadows.

Optimizations

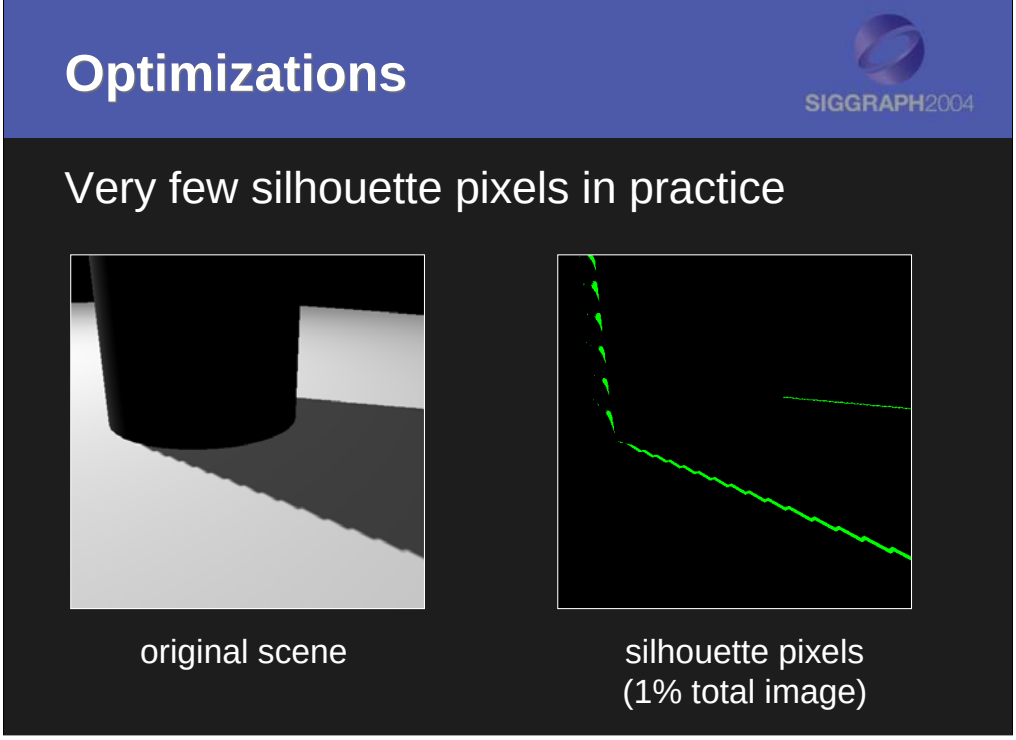
Fragment program is expensive

- lots of arithmetic
- lots of texture reads (5 silhouette points)

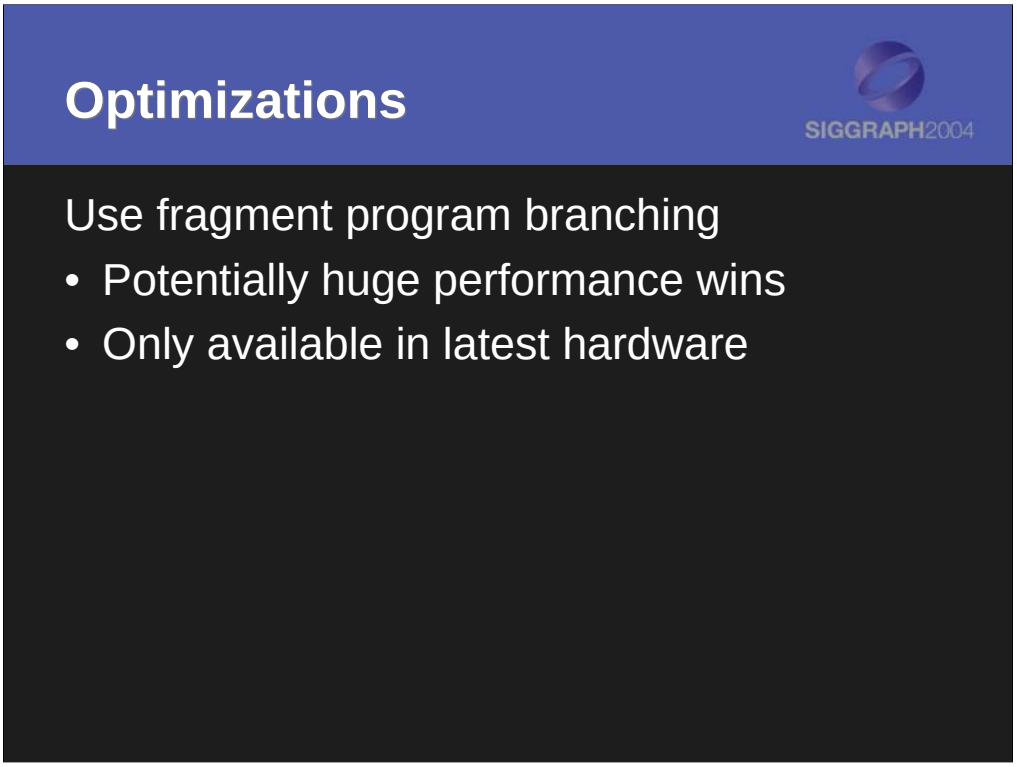
However, only required for silhouette pixels!

Now let's discuss some potential optimizations when computing the shadows. Using the silhouette map for silhouette pixels is rather expensive because it requires 5 texture reads (to gather the silhouette points) and a lot of arithmetic to perform the point-in-wedge tests. The good news, however, is that this extra work is required only for silhouette pixels.

Optimizations



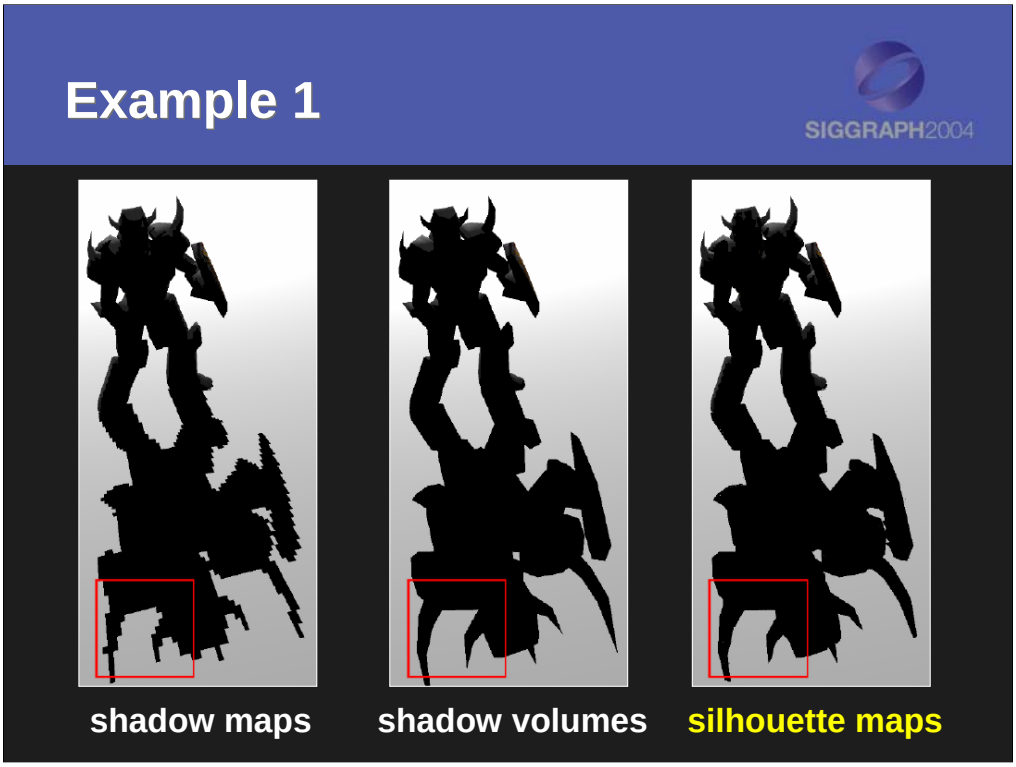
In practice, the number of silhouette pixels accounts for only a small fraction of the total number of pixels in the image. This example shows a cylinder casting a shadow onto the ground plane. The number of silhouette pixels (shown in green on the right) occupy less than 1% of the total image!



To take advantage of this observation, we simply use if/else branching in a fragment program. Most of the pixels are non-silhouette pixels, so the branching will enable us to skip the 5 texture lookups for gathering the silhouette points and all the point-in-wedge arithmetic.

Keep in mind, however, that branching in fragment programs is a very recent addition to graphics hardware, and at the time of this writing is supported only by the NVIDIA GeForce 6 series graphics cards.

Now let’s take a look at some examples and comparisons between various shadow algorithms.



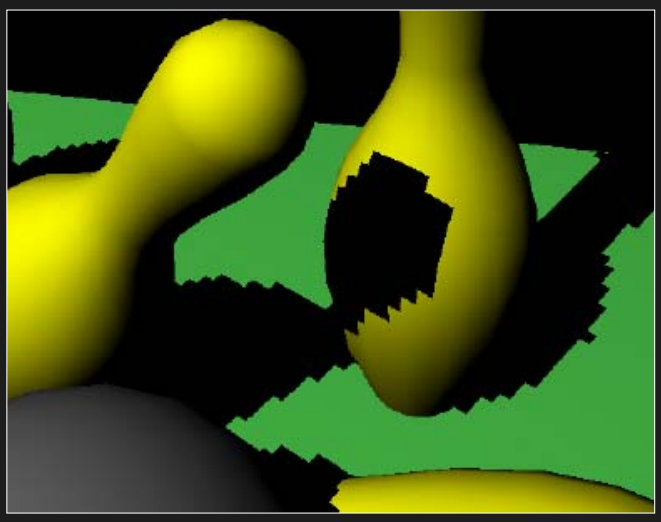
Here is a scene with the Knight character casting shadows on the ground plane. On the left is the result obtained using shadow maps. Aliasing artifacts are apparent. Shadow volumes generate accurate shadows, as shown in the middle image. The result on the right is obtained using shadow silhouette maps. Notice that it dramatically reduces aliasing artifacts, and the image is very similar to the middle image.

Example 1 (closeup)

shadow maps | shadow volumes | silhouette maps

Here's a closeup of the same scene.

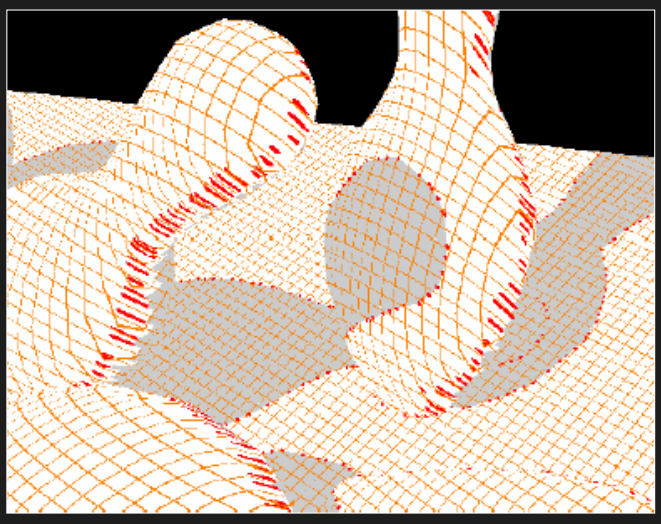
Example 2



shadow maps

Here's a second example with bowling pins (yellow) casting shadows onto each other and the ground plane. Ordinary shadow maps lead to aliasing artifacts.

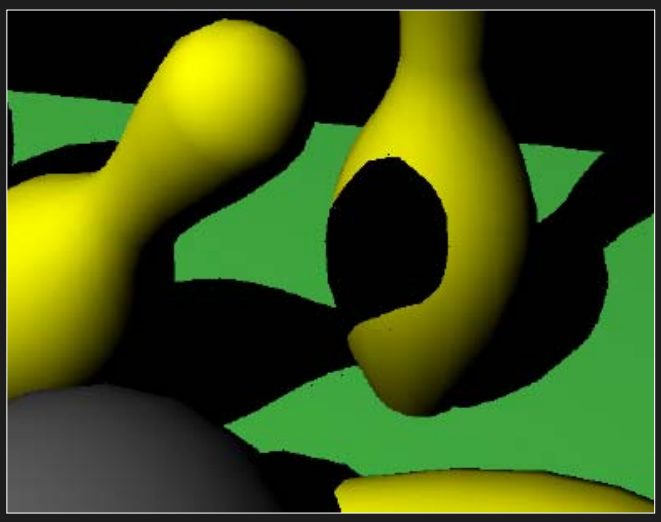
Example 2



projected silhouette map

Here's a visualization of the silhouette map, projected from the point of view of the light source onto the scene.

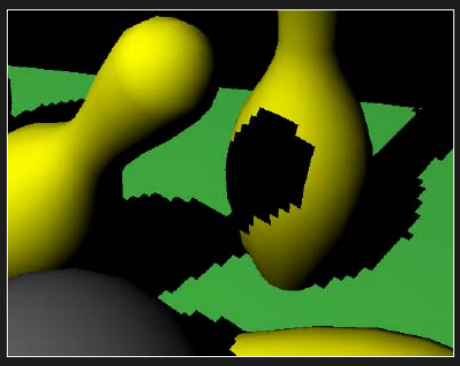
Example 2



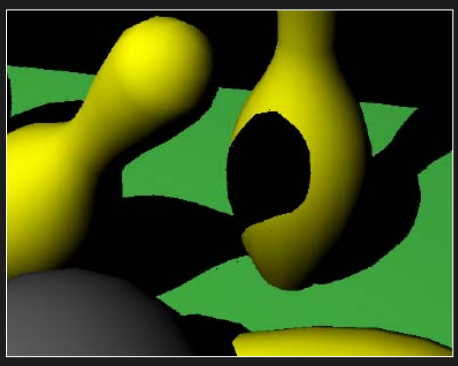
shadows using silhouette map

Here are the resulting shadows computed using a silhouette map.

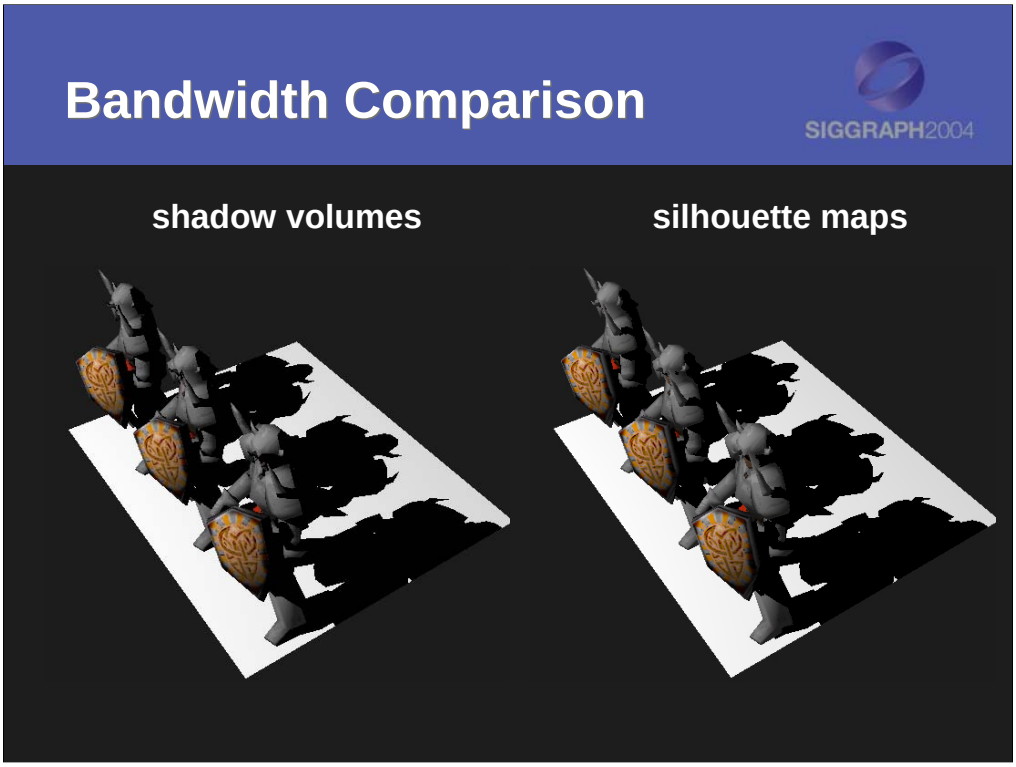
Quality Comparison



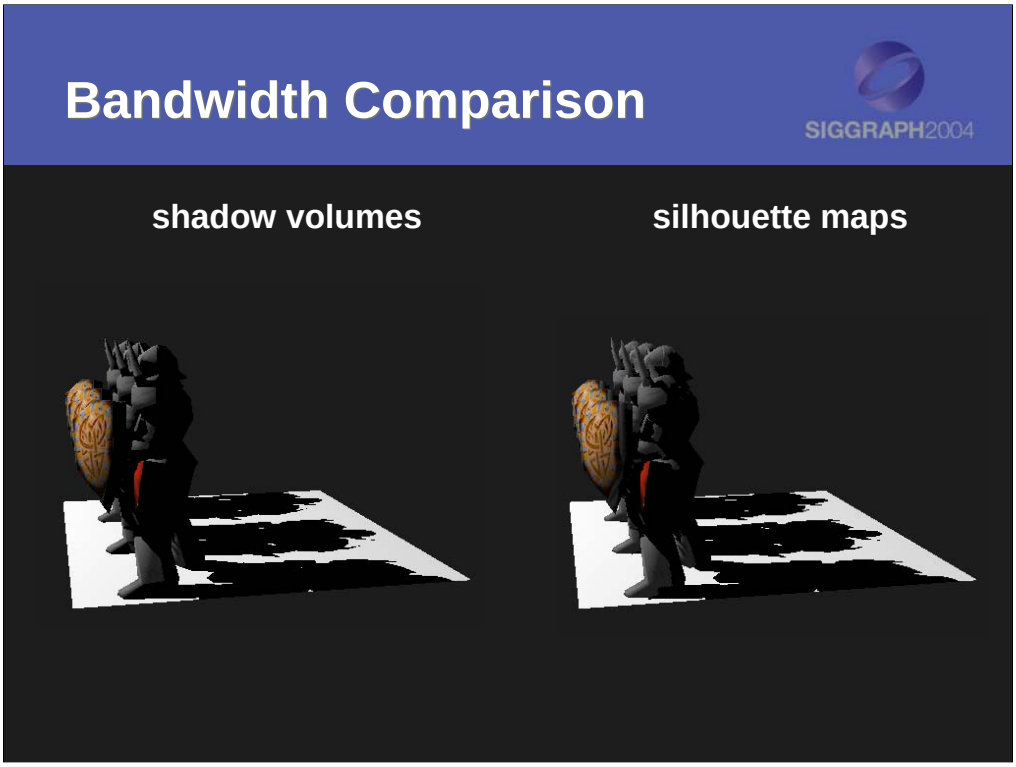
shadow map



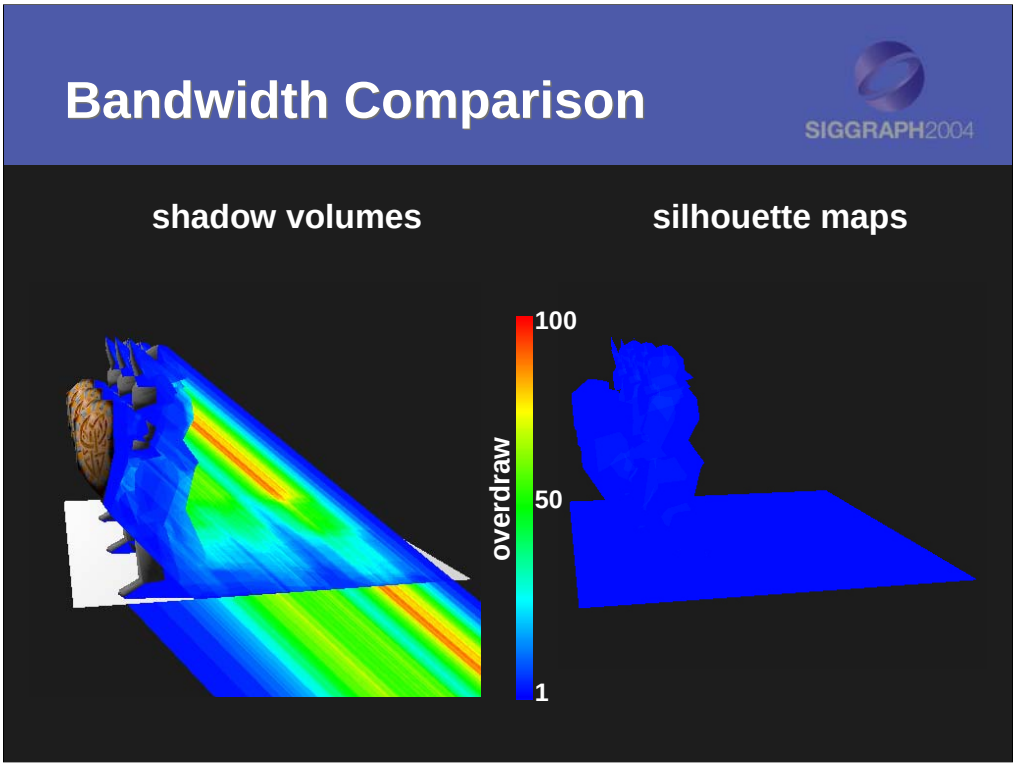
silhouette map



One of the advantages of silhouette maps over shadow volumes is that they consume far less bandwidth. Consider this example with three knights standing on the ground plane.

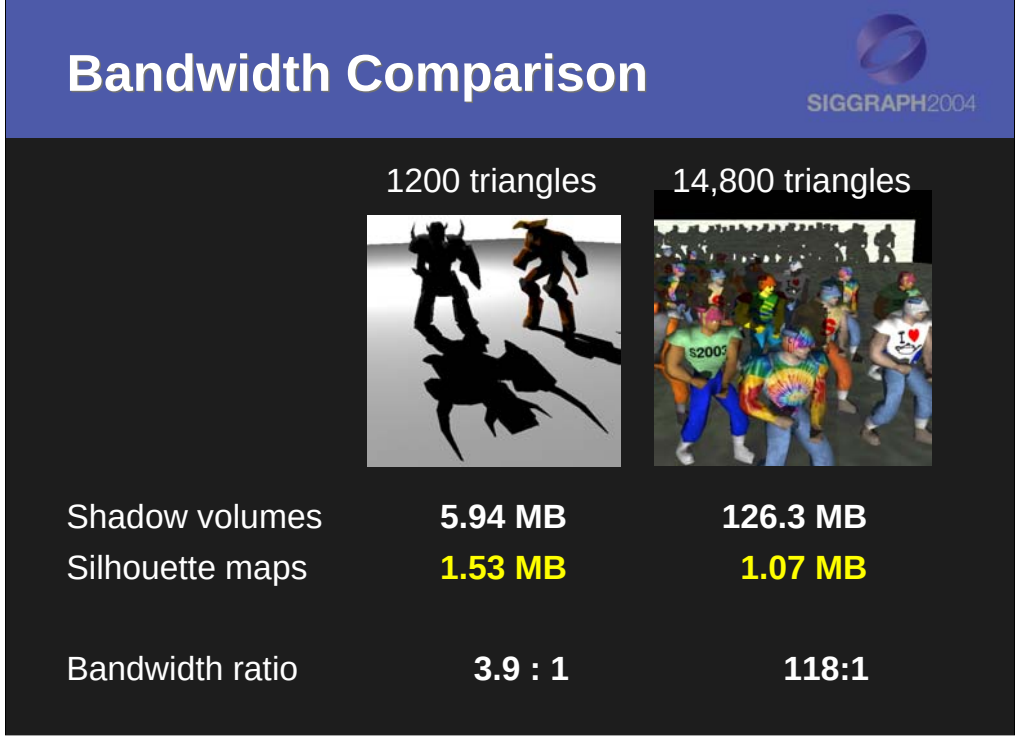


Here’s a view of the scene from the side.



Here's a visualization of the amount of overdraw when rendering the scene using the two algorithms. Dark blue regions indicate low overdraw, whereas red and yellow regions indicate high overdraw. The extra polygons in the left image show the shadow volume polygons rasterized from the observer's point of view. Clearly, the shadow volumes consume far more fillrate and bandwidth than silhouette maps.

Bandwidth Comparison



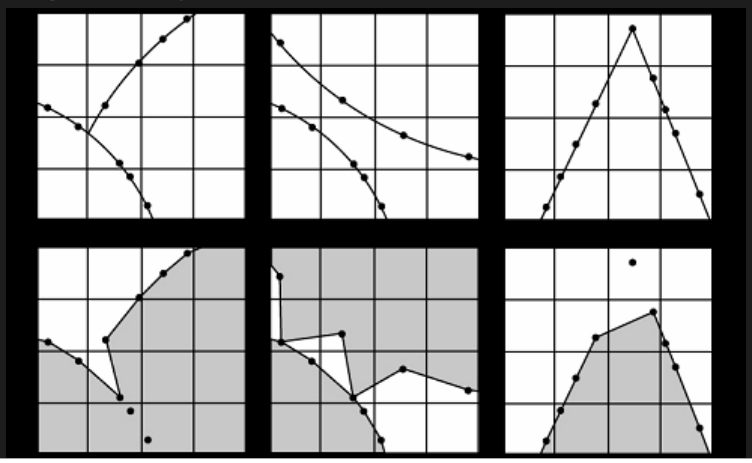
To make this more quantitative, here are two test scenes that compare bandwidth usage between the two algorithms. In both cases, silhouette maps consume far less bandwidth than shadow volumes.

Keep in mind, however, that bandwidth usage does not translate directly to performance. Shadow volumes perform many operations, but each of those operations (a stencil update) is relatively simple. In contrast, the operation performed on each pixel for silhouette maps can be rather complex. The actual performance differences between the two algorithms is highly scene-dependent and will clearly vary on a case-by-case basis.

Artifacts

SIGGRAPH2004

- Silhouette map: one point per texel
- Multiple edges inside a texel



Now let's take a look at some of the artifacts that can arise with the shadow silhouette map algorithm. We should expect to get artifacts in some cases because, after all, we are sampling the scene with a discrete buffer, limited in size by our choice of resolution for the depth map and silhouette map.

I promised earlier that I would discuss the implications of storing only one silhouette point per texel in the silhouette map. The silhouette map provides a reasonable approximation as long as only one silhouette edge passes through the texel. The main problem occurs when you have multiple, different silhouette edges that pass through the texel, as shown in the three cases above. Each column shows a different situation where artifacts can occur. In the left column, two curves meet at a T-intersection. One of the texels contains the T-intersection and the two edges, but only one point can be stored. Since there is no explicit knowledge about the T-intersections (since silhouette edges are found and rasterized independently of each other), the choice of silhouette point is rather arbitrary. In this case, a silhouette point is chosen for the lower curve, and information about the upper curve is lost. This leads to the shadow reconstruction artifact shown in the bottom-left image.

Another problematic situation is when you simply have two curves that pass near each other without touching. These may be silhouettes belonging to completely different objects. Again, texels may be crossed by two or more silhouette edges, but ultimately only one silhouette point is stored, so information about one of the curves is lost. This leads to the zig-zag reconstruction artifacts shown in the bottom-center image.

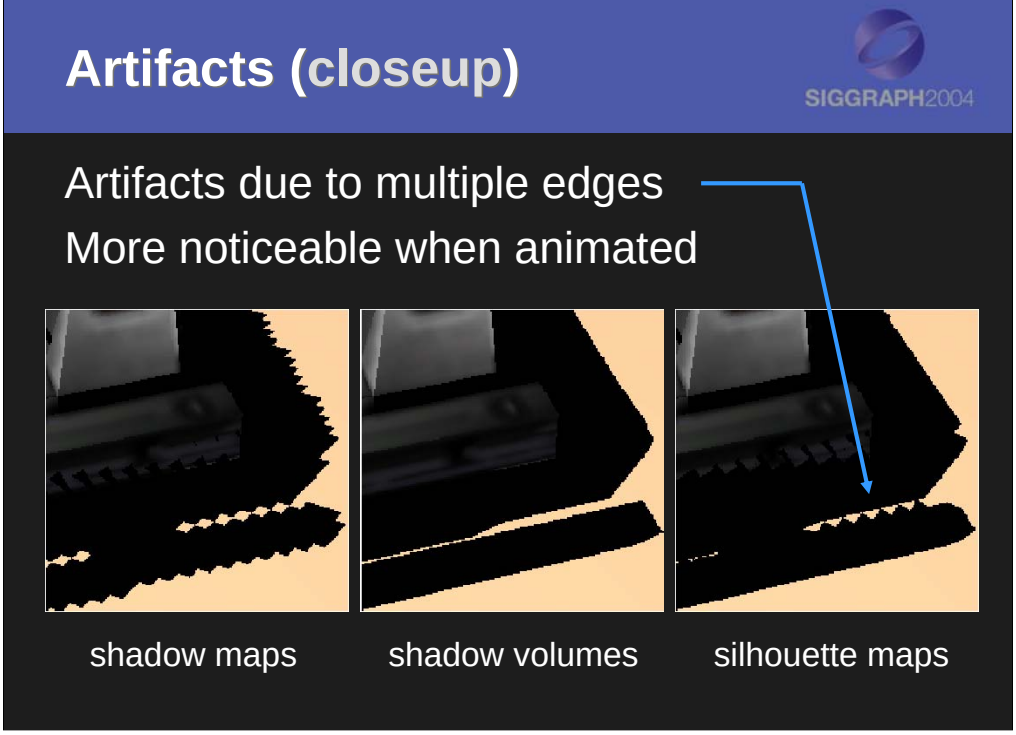
Artifacts

shadow maps | shadow volumes | silhouette maps

To summarize, the silhouette map algorithm can lead to artifacts whenever multiple silhouette edges cross a texel. Clearly this is related to the silhouette map resolution. The lower the resolution, the more likely that multiple edges will cover a given texel. Also, scenes with fine geometry tend to have silhouette edges that are close to one another.

In this jeep scene, artifacts can be seen in the shadow cast by the jeep's fender onto the ground plane.

Artifacts (closeup)



Here's a closeup of the artifacts. Note that these artifacts are generally more visible in animations, because the shadow artifacts tend to "pop" abruptly depending on how the image samples get projected onto the silhouette map. A more severe version of popping occurs with regular shadow maps.

Algorithm Comparison

Perspective Shadow Maps:

- same generality as shadow maps
- minimal overhead (2 passes)
- doesn't address aliasing in all cases

Shadow Silhouette Maps:

- addresses aliasing more generally
- more overhead (3 passes + big shaders)
- less general than shadow maps

Now that we've seen both the perspective shadow map and silhouette map techniques, let's compare these two methods qualitatively. Perspective shadow maps require minimal changes to the original shadow map method; conceptually, they just involve an extra perspective transform. Thus they require only 2 rendering passes and have the same level of generality as regular shadow maps: they automatically handle any geometry that can be represented in a depth buffer, such as polygonal models, points, sprites, and so on. However, they do not fix aliasing in all cases. In particular they do not solve projection aliasing, and they also cannot solve perspective aliasing for all scene configurations.

The shadow silhouette map algorithm fixes aliasing in a manner independent of the relationship between the light and camera perspective transforms. Thus shadow silhouette maps handle aliasing in all situations, including projection aliasing, though small artifacts remain due to undersampling. Shadow silhouette maps are also fundamentally more complicated than perspective shadow maps: they rely on fragment programs, consume more memory, and require an extra rendering pass. Additional hardware may eventually reduce this overhead. Finally, the price to be paid for the higher quality of using shadow silhouette maps is that the algorithm is less general than perspective shadow maps. Since the silhouette map algorithm requires that we explicitly find silhouette edges (in order to rasterize them), it means that we must use polygonal models. This is not a major concern for many real-time applications, since modern graphics hardware is dedicated to polygonal rendering. In the future, however, other types of primitives such as higher-order surfaces may be supported.

Combination of Algorithms

Why not combine techniques?

Perspective shadow map:

- Optimizes depth sample distribution
- More samples closer to viewer

Shadow silhouette map:

- Optimizes depth sample information
- Exact silhouette edge locations

Fortunately, shadow silhouette maps and perspective shadow maps are complementary. It is possible to combine them to get the best of both worlds, at very little added cost. Keep in mind that the two techniques address aliasing in different ways. Perspective shadow maps optimize the distribution of the depth samples in the shadow map so that more samples are assigned to regions of the image closer to the viewer. In contrast, shadow silhouette maps optimizes the amount of information provided by each sample. In particular, depth samples are effectively deformed so that they lie along silhouette edges. This explicit edge information is used to reconstruct shadow edges accurately.

Summary

- Image-space algorithm
- Silhouette map: deformed depth map
- Piecewise-linear approximation
- Scalable (compared to shadow volumes)

Compared to (perspective) shadow maps:

- Removes aliasing in more cases
- Additional overhead and requirements

There are a few key ideas to remember about shadow silhouette maps. It is an extension of the regular shadow map algorithm, and it also works in image space. In addition to the shadow map, we add a silhouette map, which conceptually helps us to represent a deformed depth map in which depth samples are located where we need them: on the blockers' silhouettes, which give rise to hard shadow edges. Since we store one point per texel in the silhouette map, we obtain a piecewise-linear reconstruction of the true silhouette curve. This looks much better than the piecewise-constant reconstruct obtained using a standard shadow map. Compared to shadow volumes, which work in object-space, the silhouette map consumes less fillrate and bandwidth and scales better to complex scenes.

Compared to existing shadow-map-based approaches like perspective shadow maps, silhouette maps offer better quality, but the tradeoff is additional overhead and less generality. This is a classic tradeoff in shadow algorithms.


SIGGRAPH2004