Implementation of Visual Selection using Shadow Volumes Technical Report

Bernhard Rainer 0828592

August 28, 2016

Contents

1 Introduction

During an internship at the *VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH* from March to June 2016, a visual selection technique has been implemented. Visual Selection was developed to preview interactive selection in large-scale point clouds. Such datasets consume several gigabytes of memory and are simply too large to be completely stored in memory, yet alone in video memory. Therefore a fast visualization needs continuous updates of the displayed data, yielding the CPU-GPU memory upload as a severe bottleneck. Given the size of modern point clouds and their scientific purpose, such datasets often have to be cleaned manually, removing unwanted parts, or extracting regions of interest for further exploration. This selection task is usually done by the user, preferably by selecting regions in 2D directly on the screen. From the camera's point of view, those regions cast a shadow, creating a volume in 3D space, in which all points are in shadow and therefore selected. A selection is a complex combination of multiple volumes, created by the user from different viewpoints. Those changes in viewpoint allow the user to capture all features of importance in one selection. A logical selection can be computed on the CPU, but the resulting changes in the dataset have to be uploaded to the GPU in order to visualize them. The combination of heavy computations and limited memory bandwidth result in a significant delay between the users' input and the display of the selection in the point cloud.

In this project, we present an efficient way to decouple the presentation of a selection on screen from the logical selection on the dataset to overcome performance issues. Logical selection and visual selection are performed in parallel. Only if a manipulation operation has been performed on a selection, an update of the GPU-memory must be performed. Since this procedure mirrors the logical selection on the CPU, both procedures must produce the same results.

The selection highlighting can be seen as a post-processing effect, where we render the selection volumes to the stencil buffer and perform a final full-screen pass to highlight selected pixel. This procedure only requires the actual scene to be rendered to the depth buffer. The visual selection works independently of the dataset, relying only on the user-provided selection volumes, making it possible to render the selection in real-time. For this project, we adapt a seemingly old technique for shadows to render the selection volumes. Shadow Volumes was developed by Franklin Crow[1] and later refined using a Stencil Buffer by Tim Heidmann[2]. The most common version of the algorithm called *depth-fail* was highly popularized by the video game Doom 3. It is also known as Carmack's Reverse, which is used for this project.

A selection is a combination of multiple volumes, thus creating a new volume. This volume is mapped to the stencil buffer as pixels with a stencil value of *one*. The creation of this combination follows a set of selection types, each manipulating the stencil buffer in a different way, in order to only mark wanted pixels with *one*. The approach of combining volumes in screen space has been done before for *constructive solid geometry*[3]. We use similar combination types, as well es new types, to combine multiple volumes. The algorithm will be explained in detail in section 2, where we highlight the key features of this implementation. Section 3 will give an introduction

on how to use the program and which frameworks are necessary for it to run. Section 4 shows the performance of the program and its real-time capabilities. Section 5 will conclude this report with an outlook on future improvements and a final conclusion.

2 Algorithm

2.1 Overview

Visual Selection works on an 8-bit stencil buffer. It requires the selectable geometry to be rendered in an earlier render pass and stored in a depth buffer. The method consists of several steps, all of which must be executed consecutively. We start by creating a volume from a 2D polygon, drawn by the user on the screen. This polygon can be interpreted as a shadow caster, whose occluded area creates a volume for the selected region. Section 2.2 explains how such a volume is created using a geometry shader.

This volume is then rendered using the *depth fail* shadow volume algorithm described in section 2.3 in order to populate the stencil buffer. The essential part of the algorithm is the combination of different volumes. This step will be explored deeply in section 2.4, where we describe the different selection types and their effect on the stencil buffer. Lastly, in order to render the selection to the screen, a highlighting pass is performed, that renders all highlighted pixel in a certain color.

2.2 Creating 3D Volume from Screen Space Polygon

The user supplies the system with 2D polygons, which are drawn on the screen, marking the selected region in screen space. All geometry occluded by this polygon will be selected. in 3D-space this region corresponds to a volume extruded from the 2D polygon in the look direction of the camera. The polygon is triangulated in order to be rendered as a homogenous area, which is called *lightcap*. For each polygon, we store its view and projection transform in order to restore its former position in world space. We can render the *lightcap* from different views by using the inverse of the provided view-projection transform as world transform for the polygon. In order to create a volume from the *lightcap*, we extrude each edge using a geometry shader. The direction of extrusion is defined by the vector from the previous view position towards each vertex. For each edge in the polygon, we extrude a quad with a certain distance, thus giving us the hull of the volume. To close the volume the *lightcap* is rendered also, as well as the *darkcap*, which closes the back of the volume. The *darkcap* is *lightcap* reversed, translated to the end of the volume.

2.3 Depth Fail Shadow Volumes

Each pixel is either inside a volume or outside. We split the volume in front and back faces and render them separately in order to apply different stencil operations. A point is inside the volume if the back face is behind the point and a front face is in front of

the point. We can make use of a stencil buffer to achieve this behavior. We render the back faces of the volume with the following stencil set up:

```
glStencilFunc(GL_ALWAYS, 0, 0xFF)
glStencilOp(GL_KEEP, GL_INCR_WRAP, GL_KEEP)
```

`glStencilFunc` describes the function of the stencil test. In this case, no pixel is discarded, since the compare function will always return true. `glStencilOp` describes the operation executed on the stencil buffer. In our case, the second parameter is of value to us. For each back face, we increment the value of the stencil buffer if the depth test fails. We render the front faces of the volume with decrement, instead of increment:

```
glStencilOp(GL_KEEP, GL_DECR_WRAP, GL_KEEP)
```

We decrement the values of the stencil buffer if the face is behind the point as well. The result of those operations is: If a back face is behind the object, the stencil value is incremented; if a front face is behind the object as well the stencil value gets decremented again, resulting in a stencil value of zero. If a front face is in front of the object the stencil operation will not come to effect. Therefore the value in the stencil buffer will be greater than zero. In practice, the different stencil functions and operations for front and back faces can be combined into one command each, so the volume can be rendered with a single draw call. If we look at a ray from the camera to an object, the ray alternately intersects back faces and front faces. This means, that the stencil buffer is incremented and decremented alternately as well. Therefore the stencil buffer is populated only with *zeros* and *ones* after a single volume is rendered, which is the desired state of the stencil buffer. Figure 1 showcases the algorithm for three objects inside and outside of a volume.

In theory depth-fail can support infinite long volumes, which are capped at the far plane. This requires an infinite perspective projection. In this project we deliberately do not use this feature, instead we give the user a controllable selection distance, which caps the volume at this distance. It's counterpart depth-pass may run faster and has less geometry to render, but problems occur, when the volume is clipped by the near plane. This leads to incorrect results when the camera is inside a volume. The depth-fail algorithm works correctly for all camera positions, especially inside volumes. Therefore the user can navigate inside a selected region for further selection and exploration.

2.4 Combining Multiple Selection Volumes

Combining multiple volumes is essential to obtain a complex selection. Whilst one volume trivially is a selection by itself, multiple selection volumes have to be combined using the different selection types **And, Or, Xor** and **Subtract**. Each selection type combines the volume rendered earlier (stored in the stencil buffer as *ones*) with a new

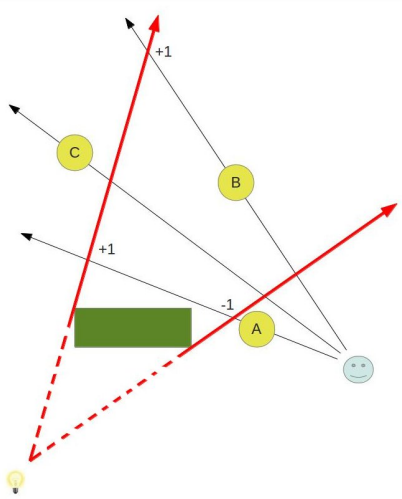

Figure 1: Objects A is in front of the shadow, B inside, C behind.

A	B	And	Or	Xor	Subtract
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Table 1: Combining A and B using different selection type results in different stencil values.

volume provided by the user. Table 1 shows the results for each selection type applied to A and B on the content of the stencil buffer.

Single, And, Or and **Xor** volumes use the same stencil set up. The stencil value is incremented by *one* if the pixel is inside the volume. SUBTRACT has the functionality of a negative volume. We do not want points inside this volume to be selected. Therefore we flip the stencil operations and decrement for back faces and increment for front faces. The volume now effectively decrements the stencil value by one if a point lies inside of it.

The desired population of the stencil buffer is *one* for selected pixel and *zero* for not selected pixel. When multiple volumes are rendered, pixels in overlapping regions are incremented or decremented multiple times. Therefore values in the stencil buffer can differ from *zero* and *one*. Table 2 shows the population of the stencil buffer after a selection volume is rendered.**Min** describes the minimum value, that may be present in the stencil buffer after the selection, **Max** the maximum value. **Selected Values** describes all stencil values, that mark selected pixels for each selection type.

Selection Type	Min	Max	Selected Values
SINGLE	0	1	1
OR	0	2	1,2
AND	0	2	2
XOR	0	2	1
SUBTRACT	-1	1	1

Table 2: Population of the stencil buffer after a volume is rendered.

Selection Type	Stencil Function	Reference	Stencil Fail Operation
SINGLE	-	-	-
OR	`GL_GREATER`	1	`GL_REPLACE`
AND	`GL_GREATER`	1	`GL_DECREMENT`
XOR	`GL_GREATER`	2	`GL_ZERO`
SUBTRACT	`GL_EQUAL`	1	`GL_ZERO`

Table 3: Different stencil setups for the reduction operation for each selection type.

Rendering multiple, possibly overlapping, volumes does not create the wanted selection yet. We can see selected pixels, are not necessarily populated with *one* anymore, vice versa for not selected pixel. In order to combine multiple selections, described in table 1, an additional step has to be executed after each volume is rendered. We call this step of the selection **Stencil Buffer Reduction**. This step is essential to the algorithm. We sieve out all stencil values, that should be marked as selected and reduce their values back to *one*, or *zero* if not selected.

2.4.1 Stencil Buffer Reduction

We are only interested in the pixels affected pixels by this volume. Therefore we render the volume a second time without depth test and a stencil setup, that reduces the values in the stencil buffer back to *zero* and *one*. Since we are only interested in the covered area in screen space, we only render the front faces of the volume. Table 3 shows the different stencil setups for the selection types. **Stencil Function** describes the test function parameter, **Reference** the reference parameter for the `glStencilFunc` function. We use 0xFF as the mask for all cases. **Stencil Fail Operation** describes the stencil fail parameter for the `glStencilOp` function. This action is only triggered when the stencil test fails. The remaining parameters are set to `GL_KEEP`.

SINGLE selection is virtually identical with OR selection, only with the difference, that SINGLE selection does not need a reduction after the volume is rendered. With these stencil setups we can effectively reduce any population of a stencil buffer, that may occur within this application, back to *zero* and *one*.

2.5 Inverting the Selection

Inverting the selection can be seen as a two-pass operation. After this step, all *ones* are set to *zero* and vice versa. We do not want to invert the whole stencil buffer tough, only pixels containing rendered geometry. We render two full-screen quads at the far plane of the camera, both with different stencil setups. In the first pass, we increase all stencil values by one if the depth test fails. This results in pixels with geometry to become *one* if not selected and *two* if selected. The second pass uses the XOR reduction to set all values larger than *one* to *zero*.

2.6 Selection Highlighting

Selection highlighting is rather undemanding. We perform a full-screen pass after all volumes are rendered, in which we color all pixels, that pass the stencil test. We test for stencil values equal one.

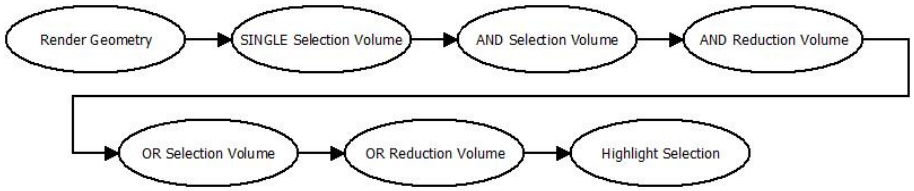

Figure 2: All render passes for an example selection consisting of SINGLE selection, AND selection, and OR selection. Finally, the selection is highlighted.

3 Usage

The program is written in F# using the Aardvark[4] framework, developed at the VRVis Research Center. The following code shows the usage of the program:

```
// Create Lasso
let lasso : Selection = ...
// Create scenegraph with geometry attached
let sceneGraph : ISg = ...


// Highlight Color
let selectionColor = Mod.constant C4f.Red
// Volume Color (Only used if showVolumes is true)
let volumeColor = Mod.constant (C4f(0.0f, 1.0f, 0.0f, 0.1
    f))
// Selection Distance
let selectionDistance = Mod.constant 5.0
```

```
// View Transform
let viewTrafo = view |> Mod.map CameraView.viewTrafo
// Projection Transform
let projTrafo = proj |> Mod.map Frustum.projTrafo
// Show Volumes or not
let showVolumes = Mod.constant false
// Last renderpass with which geometry was rendered
let geometryPass = Rendering.RenderPass.main
// Runtime & Framebuffer needed for shader compilation
let runtime = app.Runtime
let framebufferSignature = win.FramebufferSignature

// Create VolumeSelection = Scenegraph with selection and
    last render pass
let (sg, renderPass) = VolumeSelection.Init
                                          sceneGraph
                                          viewTrafo
                                          projTrafo
                                          lasso
                                          selectionColor
                                          selectionDistance
                                          volumeColor
                                          showVolumes
                                          geometryPass
                                          runtime
                                          framebufferSignature
```

The technique can be included into any project using only the `Init` method and the provided parameters. It requires an Aardvark scene graph and a Lasso as defined above. It is designed in such a way, that the procedure is updated automatically when basic values change (e.g. camera movement, color changes). Figure 3 to 7 showcase a typical selection starting with a single selection, subtracting a selection and inverting the selection.

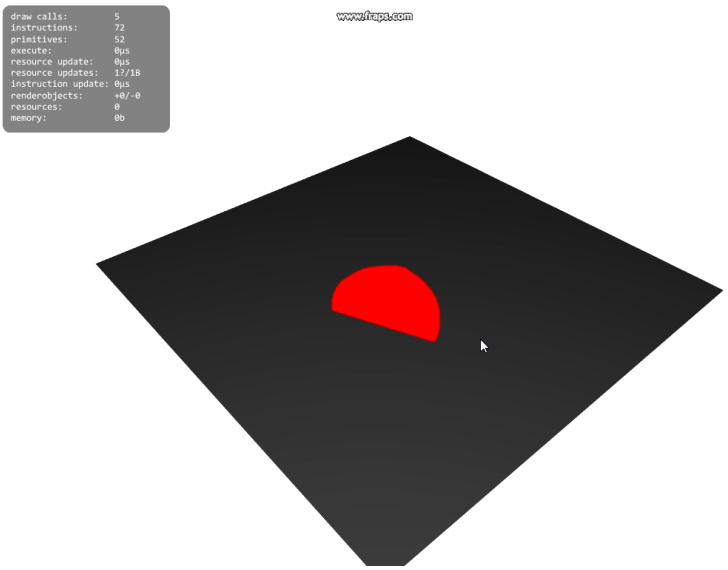

Figure 4: SINGLE selection created with the lasso above (red).

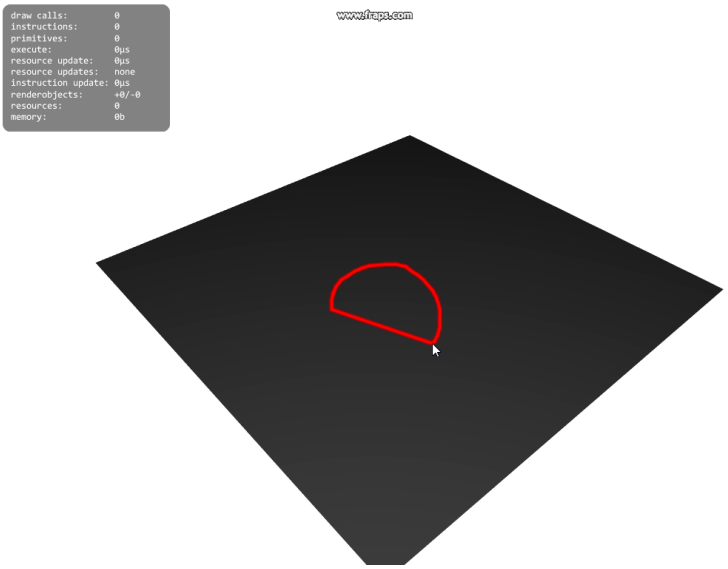

Figure 3: Lasso for selection on a test plane.

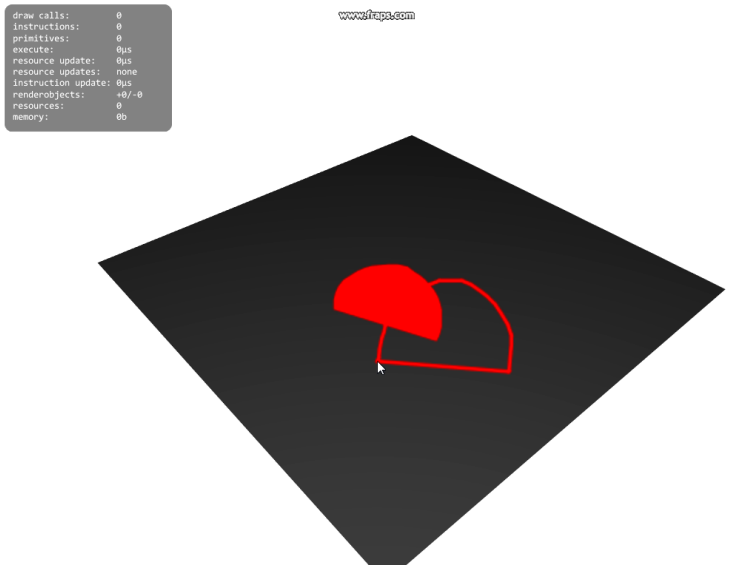

Figure 5: Lasso for the second (Subtract) selection

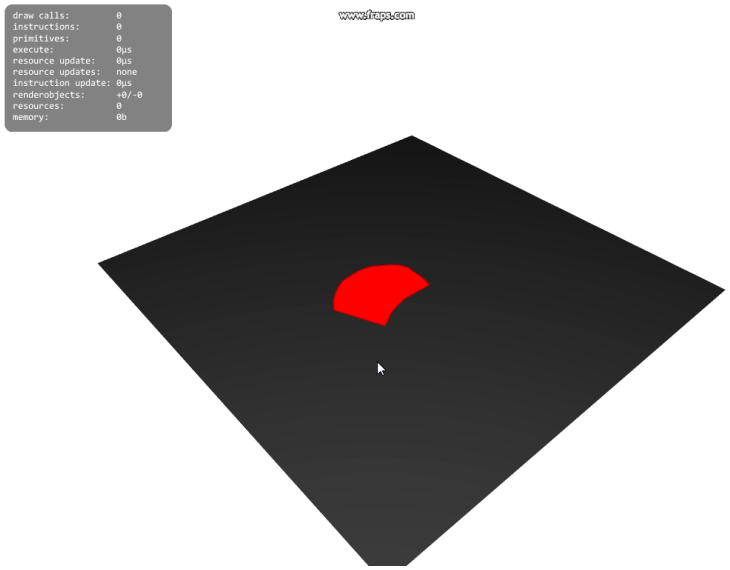

Figure 6: SUBTRACT the second selection from the first

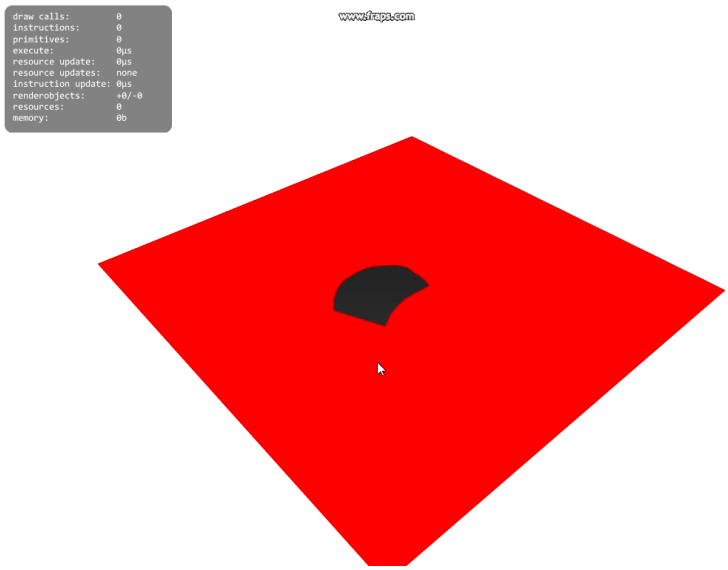

Figure 7: Inverting the selection

4 Performance

The visual selection happens purely in screen space and works independently of the dataset. Therefore the complexity directly scales with the number of selections provided by the user. The algorithms' logic is mostly performed using the stencil buffer's functionalities, thus leaving the shader code pretty lightweight, with the exception of the geometry shader. Each input triangle emits one quad, consisting of two triangles, per edge, the triangle itself and a mirrored triangles to close the volume, resulting in a total number of eight output triangles per input triangle. The vertex and fragment stage of the shaders do not process texture lookups or perform heavy lighting calculations. On the CPU, however, we must triangulate the input polygons and upload it to the GPU memory each selection update. Since those polygons are created by the user, the number of vertices, as well as the number of selections, should be filtered to be kept low.

We tested the implementation on an AMD R9 270X graphics card. Figure 8 shows the frame times per number of selection volumes. Even tough user input will most likely supply a low number of selection polygons before a manipulation is executed, the application achieved very feasible frame times for more than 200 polygons. In Figure 8 we can see a linear coherence between frame time and number of selection polygons. While benchmarking we reached an average render time per polygon of about 0.26 ms. Figure 9 shows the time per polygon for different numbers of polygons and how the number stays fairly constant for larger numbers of polygons.

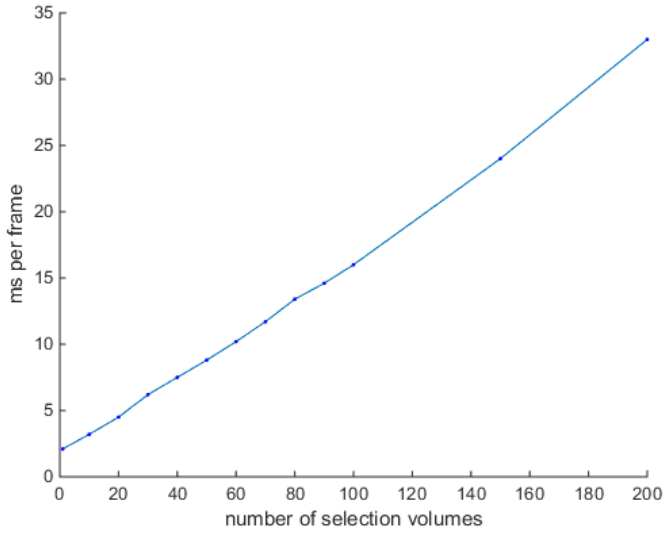

Figure 8: Average frame times per number of polygons

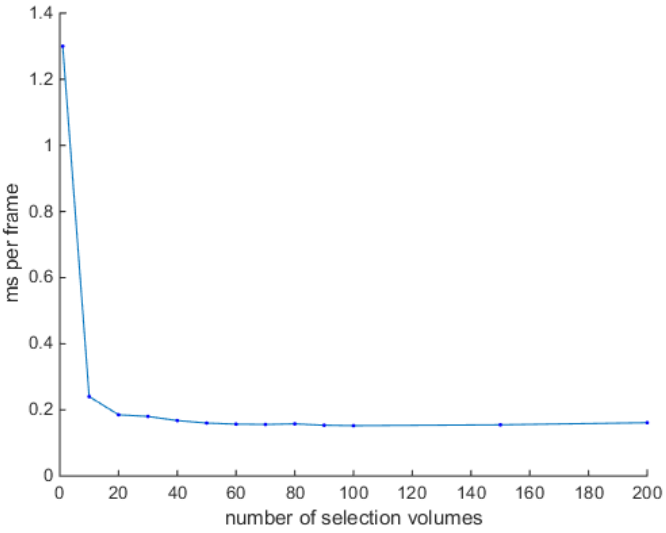

Figure 9: time per polygon for different numbers of polygons

5 Conclusion and Future Outlook

An improvement of shadow volume algorithms is to only extrude the silhouette of objects, rather than every edge of the triangulated polygon. This, however, often results in many pixel-wide holes in the selection. In some test cases and view angles, holes in the volume can be seen. This usually happens when the hull of the volume is projected onto lines only. The cause for this might be hull triangles aligned with the view direction, so they only appear as a pixel-wide line. Figure 10 shows this behavior. In this implementation, this problem disappears when the `SimpleRenderWindow` is created with a 4 or more samples for Anti-Aliasing.

Figure 10: Selection on a plane: Pixel-wide holes appear (left), but disappear when changing the camera's position (right).

As mentioned in the introduction, this method was developed for selection in large-scale point clouds, where the memory bandwidth to the GPU is a bottleneck and updates in the dataset are very costly. Point clouds are rendered using imposter spheres as geometry for each point. The visual selection only selects part of those spheres, since Shadow Volumes deliver pixel-accurate shadows. This behavior can only be suppressed, by rendering the scene again with the same stencil test as the highlight pass.

This project was included in a point-cloud visualization and manipulation application to deal with the display of the selection highlighting on the screen. Whilst this technique is very suitable for fast visualization, it does not provide methods for manipulating the dataset. This task is performed on the CPU due to the complexity of the data structure and the memory consumption of the point cloud. With this selection, the user can perform manipulative operations on the point cloud such as removing redundant information or extracting regions of the dataset for deeper exploration.

References

[1] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11(2):242–248, July 1977.

[2] T. Heidmann. real shadows real time. *Iris Universe, Number 18*, pages 28–31, 1991.

[3] Nigel Stewart, Geoff Leach, and Sabu John. A z-buffer csg rendering algorithm for convex objects. 2000.

[4] Aardvark - VRVis, Retrieved from: `http://www.vrvis.at/research/projects/aardvark/`, June 2016.