VRVis Research Center

Implementation of Visual Selection using Shadow Volumes

Technical Report

Document Version: 2016-06-21

Document Author: Bernhard Rainer

Table of Contents

1. Introduction 2

2. Algorithm 2

2.1 Creating 3D Volume from Screen Space Polygon 2

2.2 Depth Fail Shadow Volumes 2

2.3 Recursive Selection 4

2.4 Stencil Buffer Normalization and Inversion 4

2.5 Selection Highlighting 5

3. Usage 6

3.1 Performance 7

4. Conclusion and Future Outlook 7

5. Bibliography 7

# 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 for modern large-scale point cloud visualization and manipulation. 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.

An offline selection operation can be computed on the CPU, but the resulting changes in the dataset have to be uploaded to the GPU. The combination of heavy computations and limited memory bandwidth result in a significant delay between the users' input and the display of the updated point cloud.

The goal of the project was to adapt a seemingly old technique for shadows to a modern selection technique, that works independently of the dataset and displays the highlighted regions. Furthermore the technique is consistent over different camera positions.

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.* In this project we use *depth fail* as base algorithm for the selection of objects in a 3D space using only 2D input polygons.

This algorithm operates on an already rendered scene and does not need a second render pass of the selectable geometry. That being said, this selection technique only works in screen space and does not provide functionalities to manipulate the underlying dataset. 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 will conclude this report with an outlook on future improvements and a final conclusion.

# Algorithm

The algorithm works on an 8-bit stencil buffer and requires the selectable geometry to be rendered in an earlier render pass and stored in a depth buffer. Furthermore, the on-screen drawing of polygons is provided by the framework.

## Creating 3D Volume from Screen Space Polygon

The user supplies the system with 2D screen-space polygons, which are drawn on screen. The polygon needs to be 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 the inverted lightcap, extruded to the end of the volume.

## Depth Fail Shadow Volumes

The depth-fail algorithm works on all camera positions, making it perfect for in-depth exploration of point-clouds. The depth-fail method works as follows. Each point 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 always returns true. glStencilOp describes what should happen with 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 0. 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 0.

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 in one pass.

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 so the stencil buffer is populated with zeros and ones only. Figure 1 showcases the algorithm for three objects inside and outside of a volume.

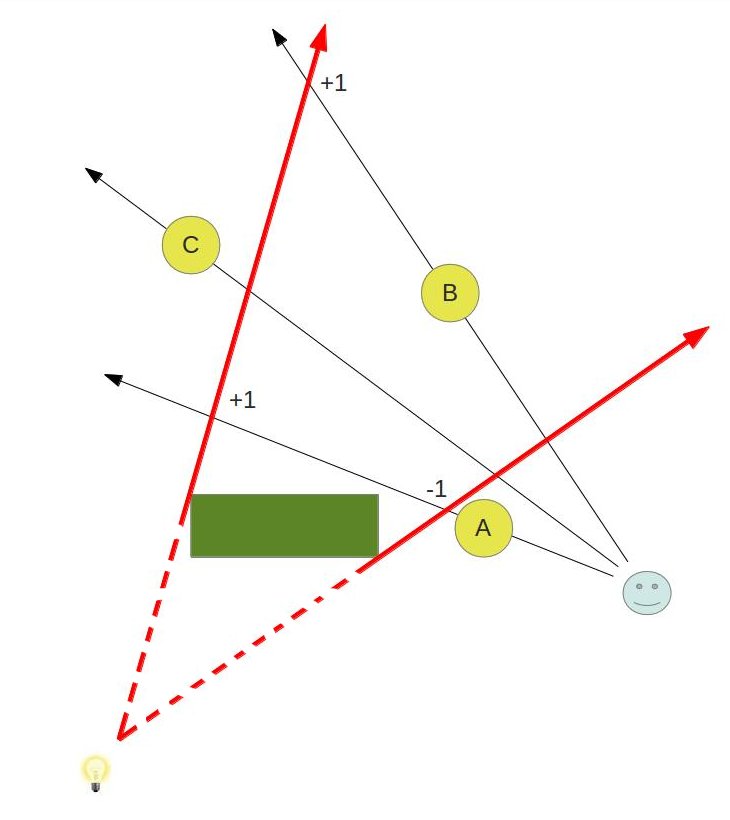


Figure 1: <http://ogldev.atspace.co.uk/www/tutorial40/tutorial40.html>

## Recursive Selection

For this project we want to apply several combinations of selections. A selection is a combination of a previous selection with a polygon. Selection types can be:

type NearPlanePolygon =

{

Polygon : Polygon2d

World2NearPlane : Trafo3d

Proj : Trafo3d

View : Trafo3d

}

type Selection =

| Single of NearPlanePolygon

| And of Selection \* NearPlanePolygon

| Or of Selection \* NearPlanePolygon

| Xor of Selection \* NearPlanePolygon

| Subtract of Selection \* NearPlanePolygon

| Invert of Selection

| NoSelection

The data structures allow us to recursively process each selection and combine it with the previous selection. Since those operators are not commutative, the order of the processed selection is of matter. For each selection polygon we then render the volume to the stencil buffer.

SINGLE, OR, XOR are additive operators. We can use standard depth-fail stencil operations. SUBTRACT however, has the functionality of a negative volume, in which we do not want points to be select. Therefore we have to invert the stencil operations and decrement for back faces and increment for front faces. Since we now render multiple volumes into one stencil buffer the values differ from zero and one. In order to simplify following selections a normalization step must be made after the selection has rendered.

## Stencil Buffer Normalization and Inversion

Our goal is to provide a stencil buffer where *one* indicates selected points and *zero* not selected points. We call this state normalized. Since values can differ from zero and one, we have to insert a normalization pass after a volume is rendered. The area of effect of each volume is the volume itself. Therefore we render the volume again, this time with a stencil setup that normalizes the stencil buffer to ones and zeros. The different selection types require different normalization steps. Table 1 shows the different normalization operations for the different selection types. **Min** describes the minimum value, that can be present in the stencil buffer after the selection, **Max** the maximum value. All stencil values, that pass the **Normalize Function** indicates selected pixels. This value will be set to one after this normalization step. **Normalize Operation** describes the operation that is executed using the stencil buffer functionalities, in order to normalize the stencil buffer.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Min** | **Max** | **Normalize Function** | **Normalize Operation** |
| SINGLE | 0 | 1 | =1 | No normalization |
| OR | 0 | 2 | > 0 | Values >1 are decremented |
| AND | 0 | 2 | =1 | Values >0 are decremented |
| XOR | 0 | 2 | =1 | Values >1 are set to 0 |
| SUBTRACT | -1 | 1 | =1 | Values < 0 are incremented |

Table 1: Normalization operations for the types of selection

SINGLE and OR selection are virtually equivalent, with the exception, that SINGLE selection does not need a normalization step afterwards.

Inverting the stencil buffer can be seen as two-pass normalization. After this step all ones are set to zeros and values of pixel, where geometry was rendered and the stencil value was set to zero, are set to one. This way, the stencil values only get inverted on pixels, with rendered geometry. We perform a full-screen pass twice. In the first pass we render a full-screen quad at the far plane and increase stencil values by one, if the depth test fails. This results in pixels with geometry to become ones if not selected and twos if selected. The second pass uses the XOR normalization to set all values larger than 1 to zero.

## Selection Highlighting

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

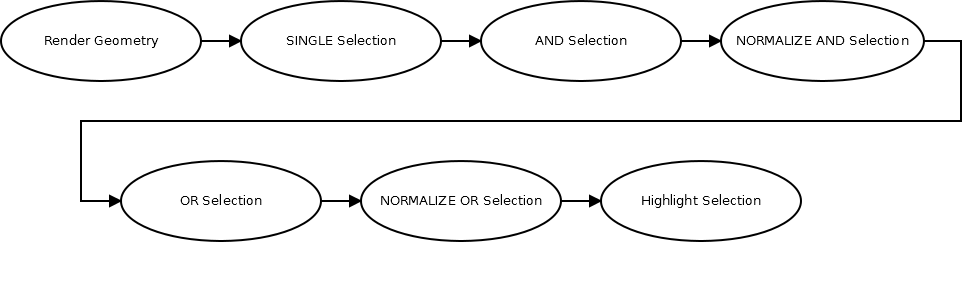


Figure 2: All render passes for an example selection of single selection combined with an AND selection, combined with an OR selection. Finally the selection is highlighted. Each node corresponds to a different render pass.

# Usage

The program is written in F# using the *Aardvark* [3] 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

let volumeColor = Mod.constant (C4f(0.0f, 1.0f, 0.0f, 0.1f))

// 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

// Renderpass with which geometry was rendered last

let geometryPass = Rendering.RenderPass.main

// Runtime

let runtime = app.Runtime

// Framebuffer

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 updates itself, if a basic value changes (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.

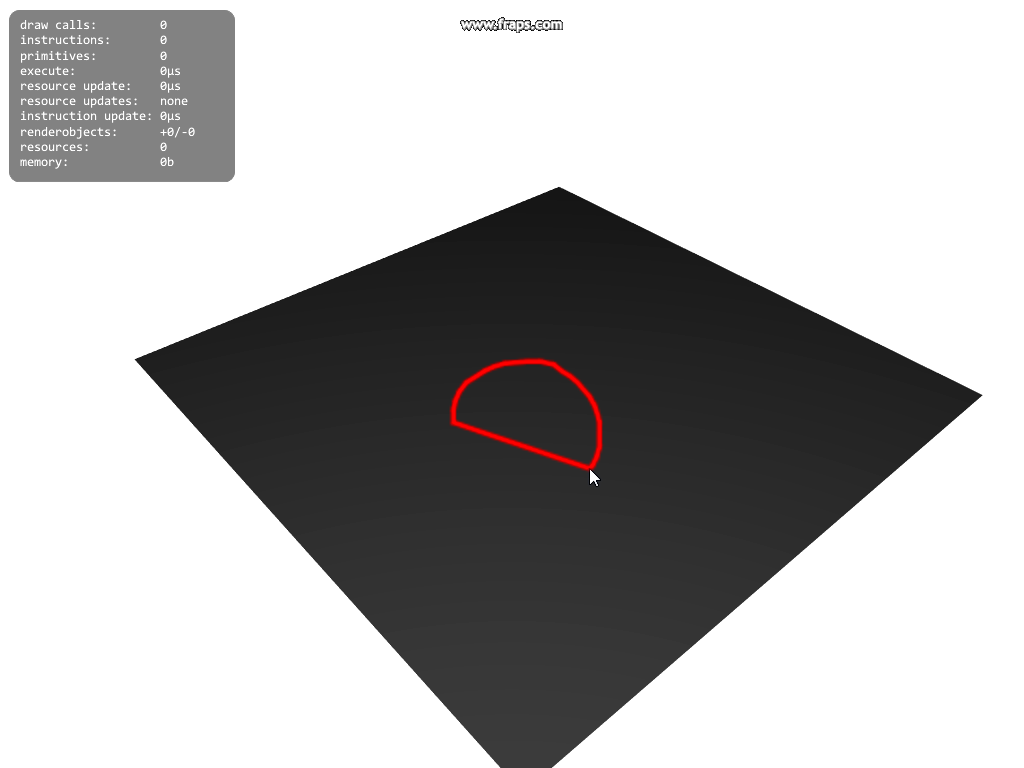


Figure 3: Lasso for selection on a testplane

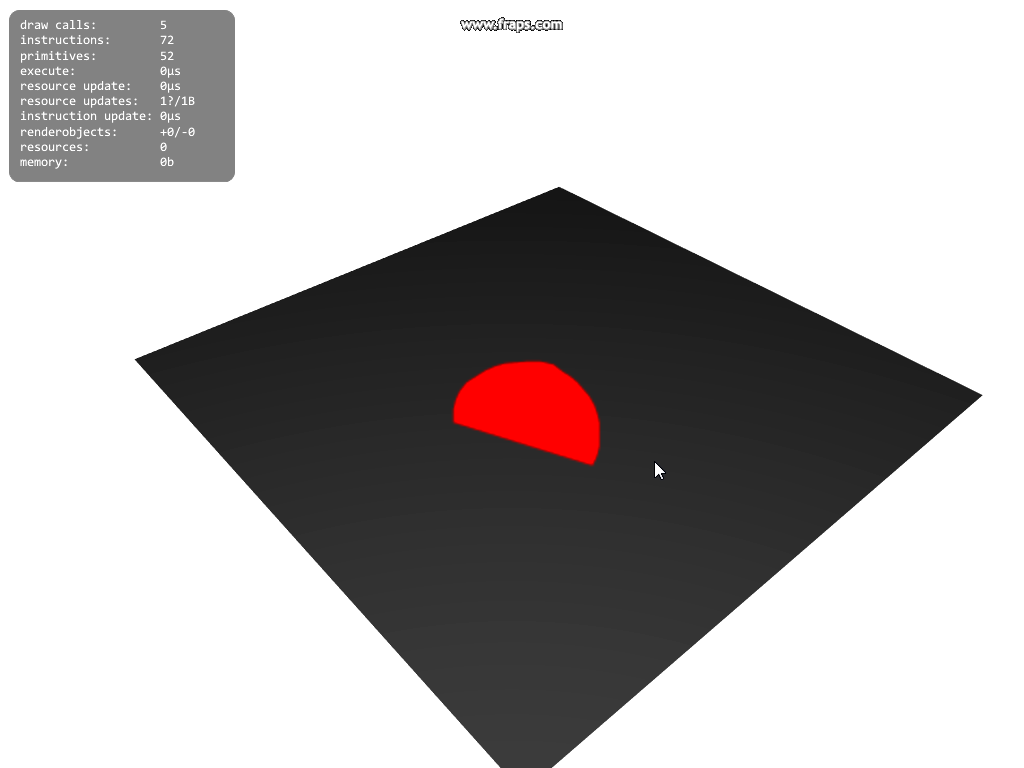


Figure 4: SINGLE selection created with the lasso above (red)

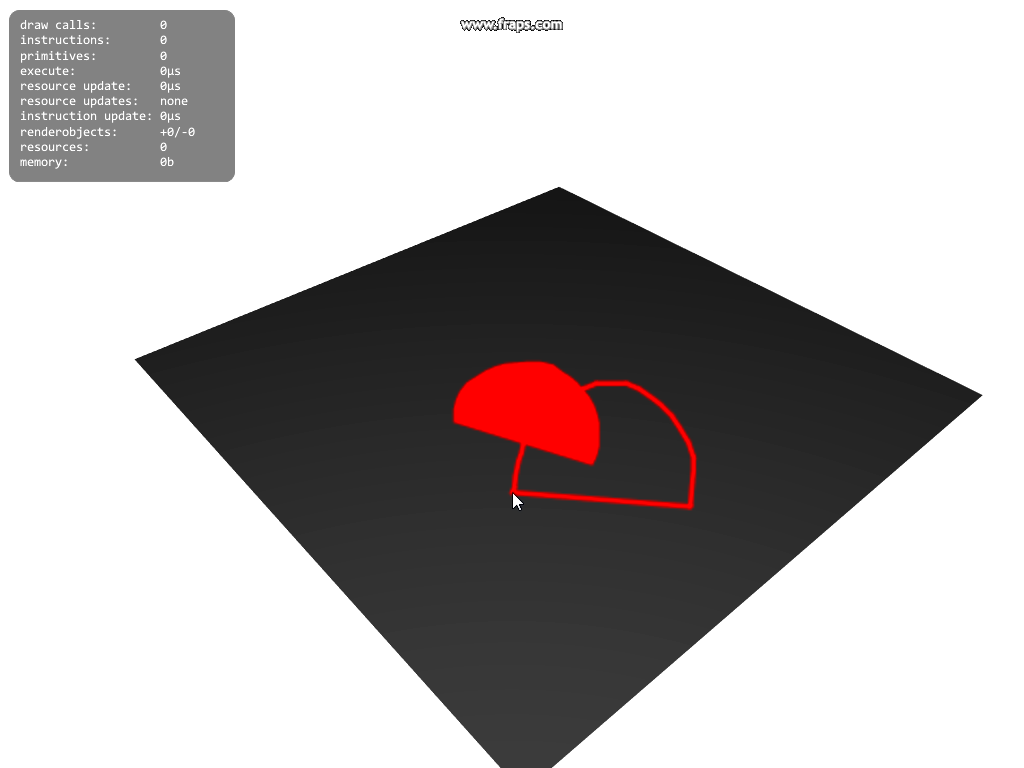


Figure 5: Lasso for second selection.

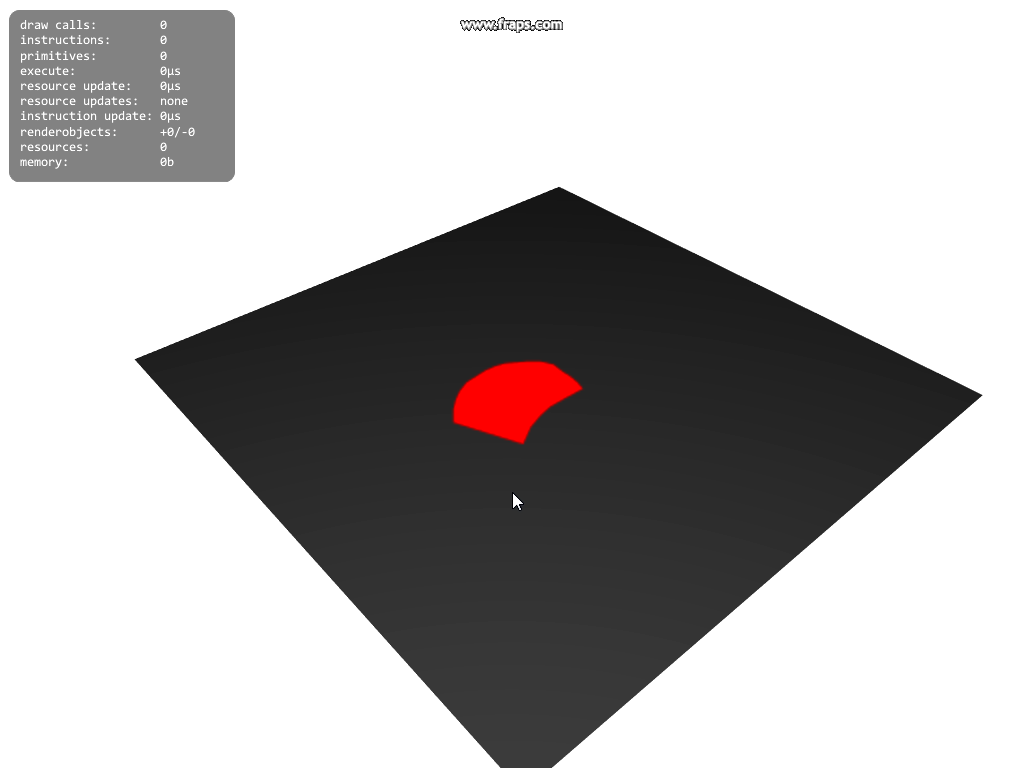


Figure 6: SUBTRACT the second selection from the first

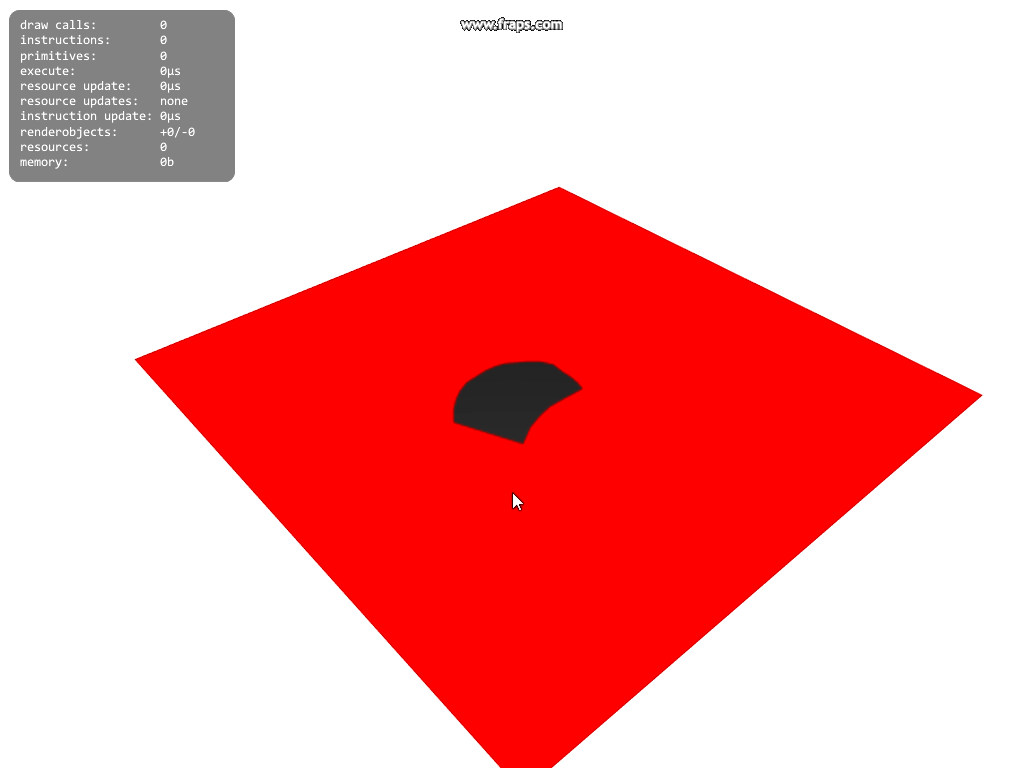


Figure 7: Inverting the selection

## Performance

The goal of this implementation was to run the selection independently of any dataset. Thus its complexity only relies on the number of polygons provided by the user. The shaders are pretty lightweight, since they do not process any texture look-ups or perform heavy lighting calculations. The geometry stage is the most expensive step on the GPU.

On the CPU however we must triangulate the lasso polygons and upload it to the GPU memory each 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.

# 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 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 8 shows this behavior. In this implementation this problem disappears when the SimpleRenderWindow is created with a 16 samples for Anti-Aliasing.



Figure 8: Selection on a plane: Pixel wide holes appear (left), but disappear when changing the cameras position (right).

Shadow volumes deliver per pixel accurate shadows. This project was implemented for selection in large Point Cloud Renderings, where the memory bandwidth to the GPU is a bottleneck and therefore updates in the selection are very costly. Point Clouds are rendered using spheres as geometry for each point. The visual selection only selects part of those spheres, since it is pixel accurate.

# Bibliography

|  |  |
| --- | --- |
| [1] | F. Crow, "Shadow Algorithms for Computer Graphics," *SIGGRAPH Comput. Graph.,* pp. 242--248, 1977. |
| [2] | T. Heidmann, "Real shadows real time," *Iris Universe 18, Number 18,* pp. 28-31, 1991. |
| [3] | "Aardvark - VRVis," [Online]. Available: http://www.vrvis.at/research/projects/aardvark/. [Accessed 29 June 2016]. |