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 3

2.1 Overview 3

2.2 Creating 3D Volume from Screen Space Polygon 3

2.3 Depth Fail Shadow Volumes 3

2.4 Recursive Selection 5

2.5 Stencil Buffer Reduction and Inversion 5

2.6 Selection Highlighting 6

3. Usage 7

3.1 Performance 11

4. Conclusion and Future Outlook 11

5. Bibliography 12

# 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 deeper exploration. Those are just two examples of manipulating operations. For such tasks, the user has to select parts of the point cloud, preferably by drawing the bounds of the region he is interested in on the screen. An offline 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 updated point cloud.

The goal of the project was to overcome those performance issues by creating a technique, that is able to highlight selected regions in a 3D environment in real-time and that works independently of the underlying dataset. In order to deeply explore point clouds, excessive changes in viewpoint might be necessary. Therefore the technique also has to be consistent over different viewpoints, so that the selected regions do not change when the camera changes.

The user draws a polygon on the screen to select geometry in the scene. The selected region can be interpreted as the shadow cast by the polygon using the camera as a light source. For this project, we adapt a seemingly old technique for shadows to define such regions.

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 the base algorithm to fill the Stencil Buffer with information on which pixel is in shadow and therefore is selected.

This algorithm operates on an already rendered scene and does not need a second render pass of the selectable geometry, like classic Shadow Volumes. The technique only stores information of the selected region and transform, making it easy to render the volumes from different viewpoints. That being said, this selection technique works as post-processing effect 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

## Overview

Visual Selection 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. The on-screen drawing of polygons is provided by the framework. The method consists of several steps, that are 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 describes a shadow volume. Section 2.2 shows how such a volume is created. This volume is then rendered using the depth fail shadow volume algorithm described in 2.3. The selection uses a recursive data structure so that multiple selections can be created easily. Section 2.4 describes this in depth. Rendering multiple volumes yields the problem, that the stencil buffer is populated with unwanted values, leading to wrong results. Section 2.5 describes the process of cleaning the stencil buffer after each selection in order to achieve a stencil buffer, that is populated with ones and zeros only. It also describes an efficient way how to invert the selection. Lastly, in order to render the selection to the screen, a highlighting pass is performed, that renders all highlighted pixel in a certain color.

## 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 the inverted lightcap, extruded to the end of the volume.

## Depth Fail Shadow Volumes

The depth-fail algorithm works for all camera positions, even positions inside volumes, making it perfect for in-depth exploration of point-clouds. The 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 operation is 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.

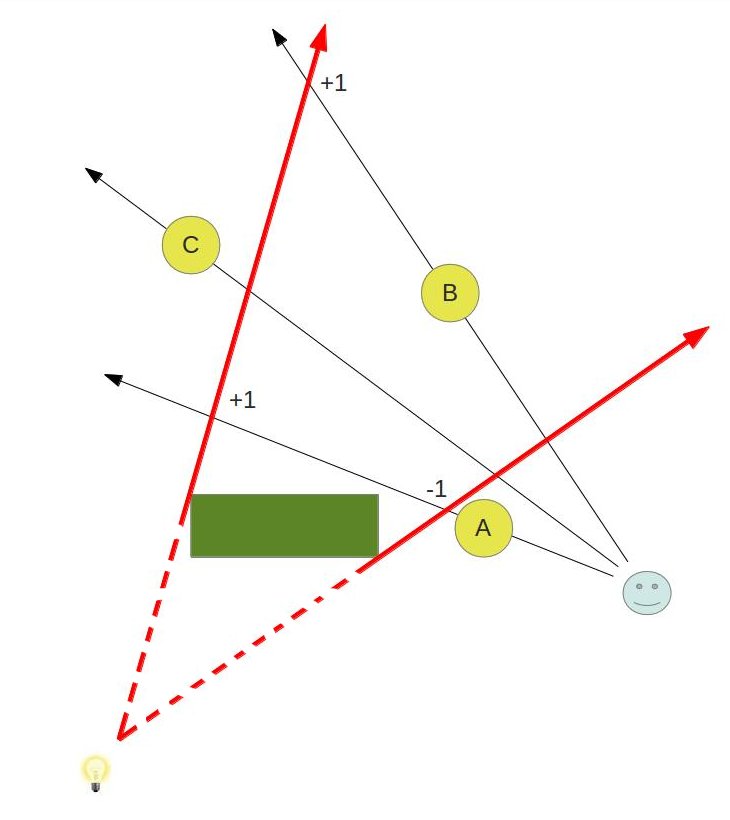


Figure 1: <http://ogldev.atspace.co.uk/www/tutorial40/tutorial40.html>

## 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. We recursively iterate over all selections, starting with the most recent and render each 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. 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.

## Stencil Buffer Reduction and Inversion

When multiple selections are processed, pixels in overlapping regions are incremented or decremented multiple times. Therefore values in the stencil buffer can differ from *zero* and *one*. This effects future selections in a negative way, such that pixels, that should be selected are not necessarily populated with *one* anymore, vice versa for not selected pixel.

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 Value** describes the stencil values, that mark selected pixels for each selection type.

To overcome this problem, the stencil buffer has to be reduced back to a valid state. Before a new selection can be applied, the stencil buffer must be populated only with *zeros* and *ones*. To achieve this we insert an additional render pass after a volume is rendered. The affected region of a volume is the volume itself. 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 *zeros* and *ones*.

|  |  |  |  |
| --- | --- | --- | --- |
| **Selection Type** | **Min** | **Max** | **Selected Value** |
| SINGLE | 0 | 1 | 1 |
| OR | 0 | 2 | 1,2 |
| AND | 0 | 2 | 2 |
| XOR | 0 | 2 | 1 |
| SUBTRACT | -1 | 1 | 1 |

Table 1: Population of the stencil buffer after a volume is rendered

The different selection types require different stencil setups. shows the operations for the different selection types. **Reduce Operation** describes the stencil set up, in order to reduce the stencil buffer back to *zeros* and *ones* for the different selection types.

|  |  |
| --- | --- |
| **Selection Type** | **Reduce Operation** |
| SINGLE | No reduction |
| OR | Values > 1 are decremented |
| AND | Values > 0 are decremented |
| XOR | Values >1 are set to 0 |
| SUBTRACT | Values < 0 are incremented |

**Table 2: Reduce operations for the different types of selection**

SINGLE and OR selection are logically equivalent, with the exception, that SINGLE selection does not need a reduction step afterward.

Inverting the stencil buffer can be seen as two-pass operation. After this step, all *ones* are set to *zeros* and values of pixels, where geometry was rendered and the stencil value was *zero*, are set to *one*. This way, the stencil values only get inverted on pixels, with 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 ones if not selected and twos if selected. The second pass uses the XOR reduction operation to set all values larger than *one* to *zero*.

## 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.

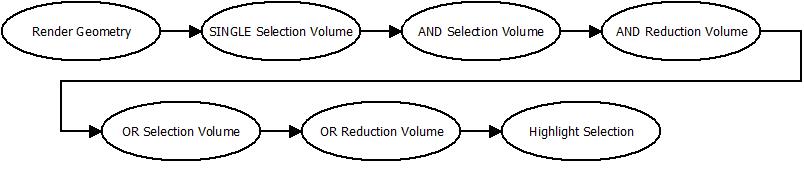


Figure 2: All render passes for an example selection of single selection combined with an AND selection, combined with an OR selection. Finally, the selection is highlighted. Each node corresponds to a different render pass.

# 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 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.

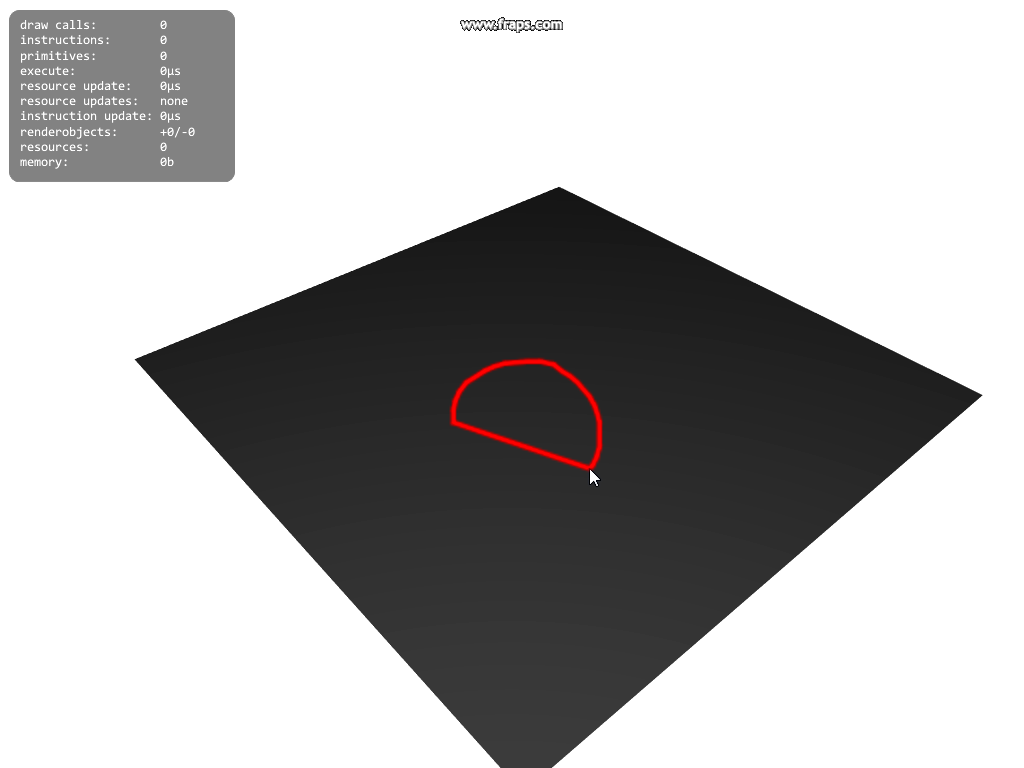


Figure 3: Lasso for selection on a test plane

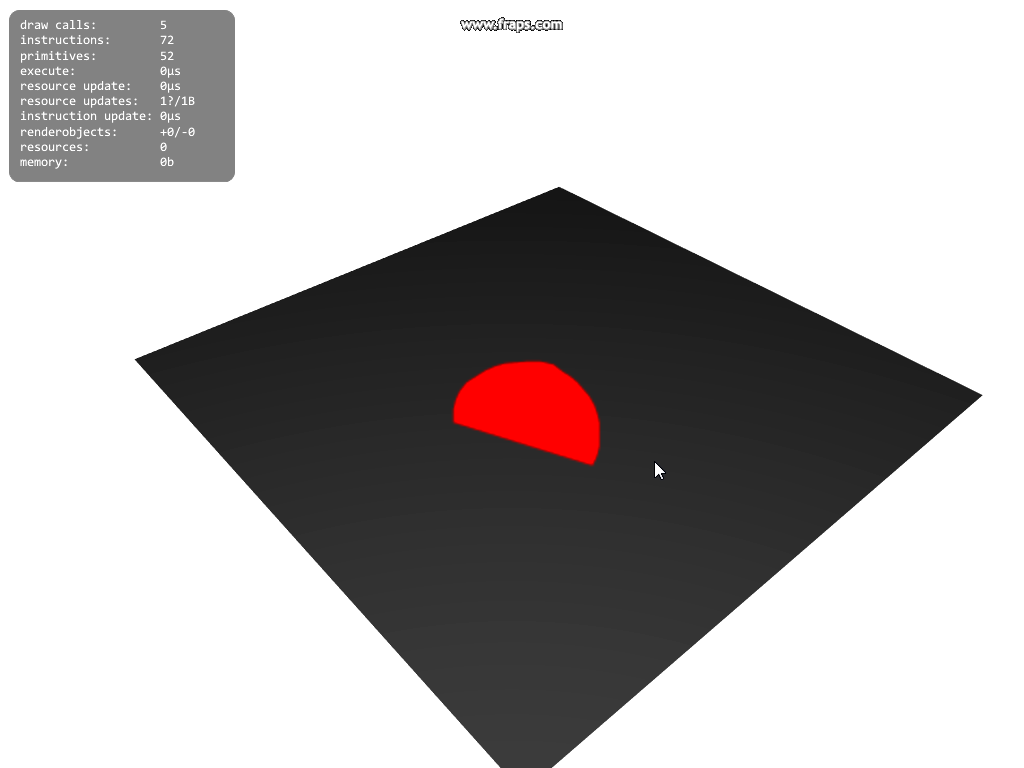


Figure 4: SINGLE selection created with the lasso above (red)

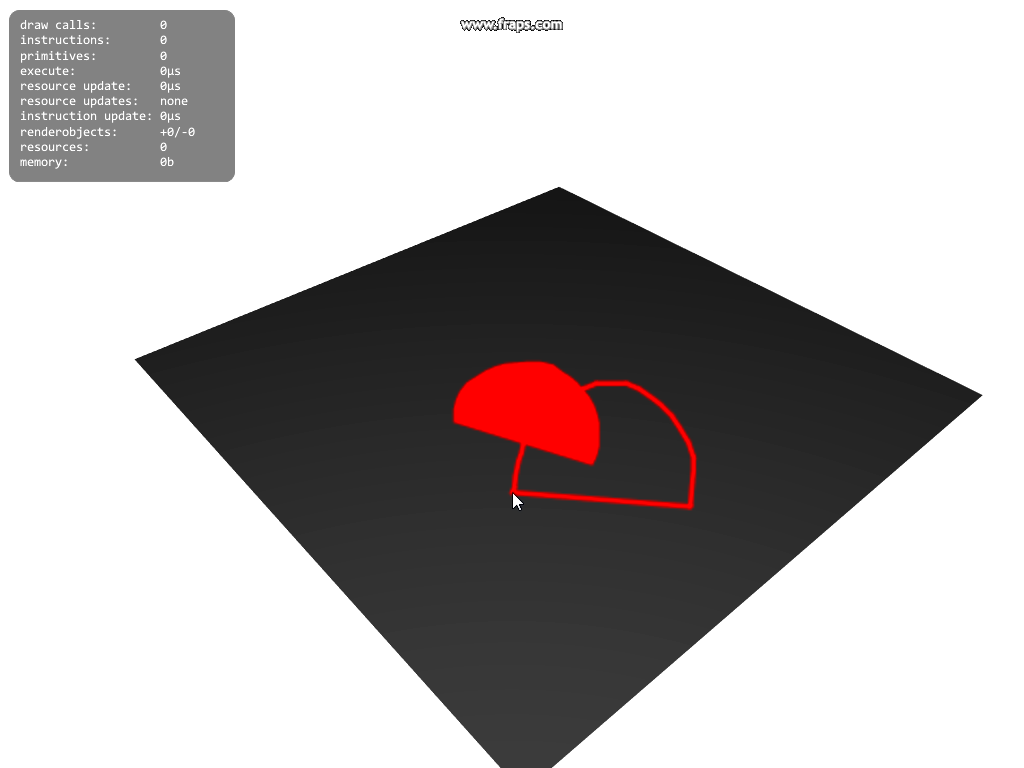


Figure 5: Lasso for the second selection.

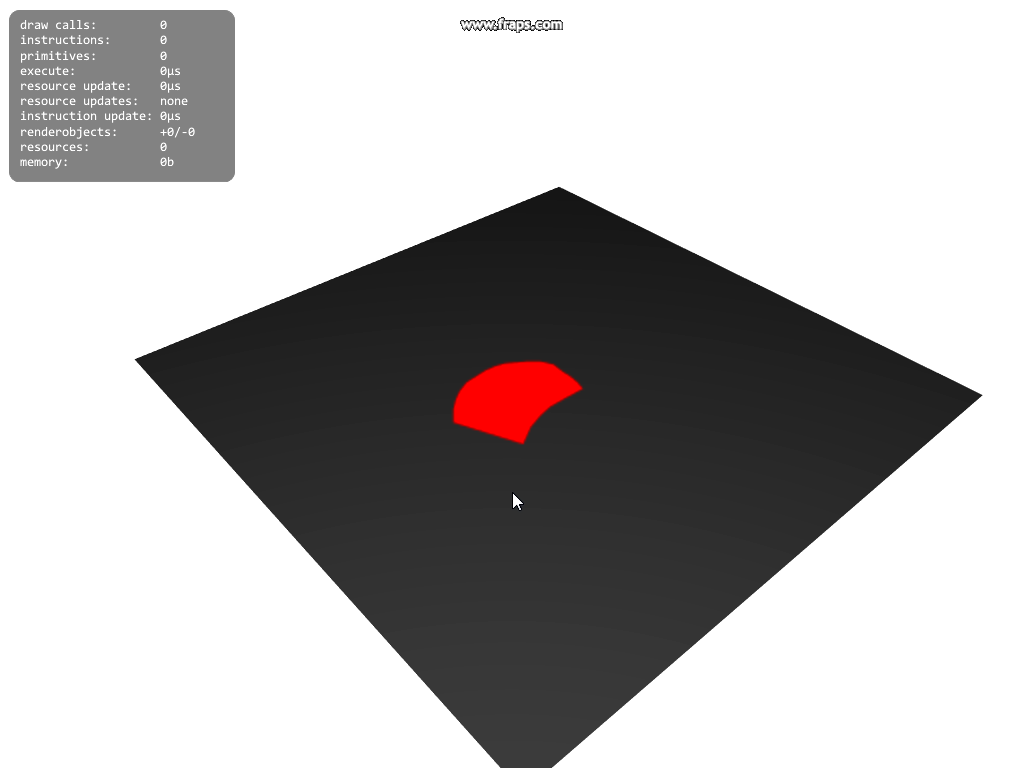


Figure 6: SUBTRACT the second selection from the first

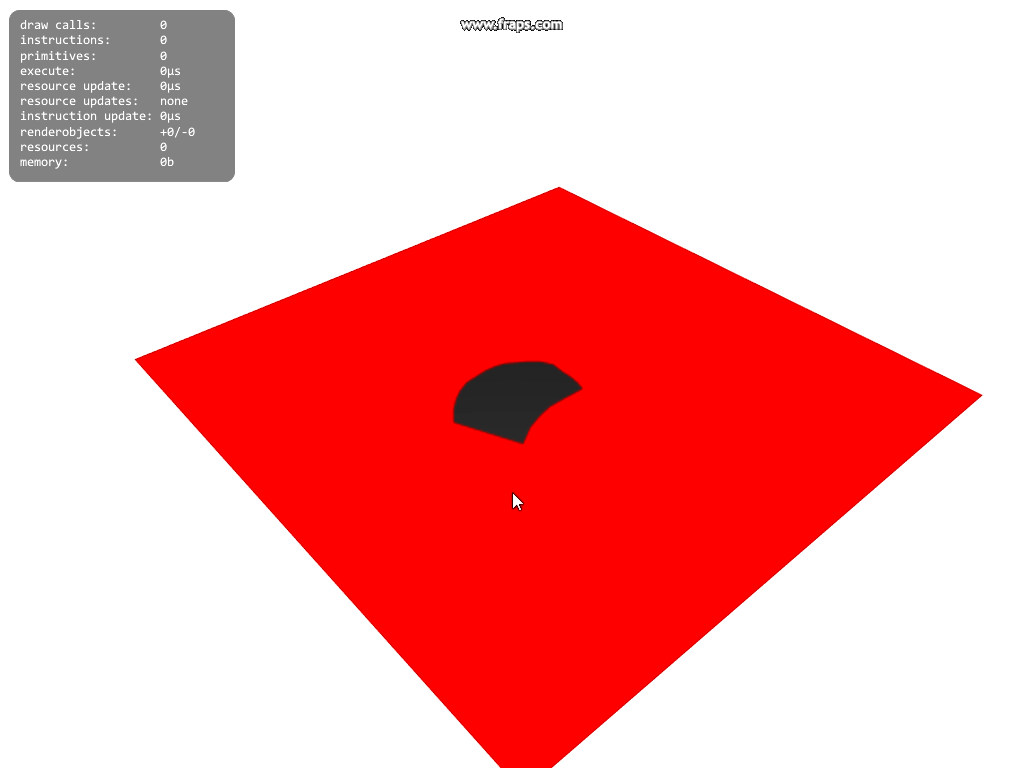


Figure 7: Inverting the selection

## 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.

# 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 8 shows this behavior. In this implementation, this problem disappears when the SimpleRenderWindow is created with a 4 or more samples for Anti-Aliasing.



Figure 8: 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.

# 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]. |