Building a Dynamic 3D Mouse Cursor

Abstract

This tutorial guides you through creating an advanced, fully customizable 3D mouse cursor implemented in C++ using OpenGL. The cursor features dynamic scaling, colour transparency, multiple geometric models, alternative “On Geometry” drawing and settings for matching user preferences. It's designed to enhance user interaction and provide contextual information in 3D space especially for navigation on a stereoscopic monitor. The code snippets provided are partially used in the 3D Cursor implementation of the Vish Program used for viewing large Point Clouds. Most of the provided code is only meant for demonstration purposes and cannot be used without further adjustments.

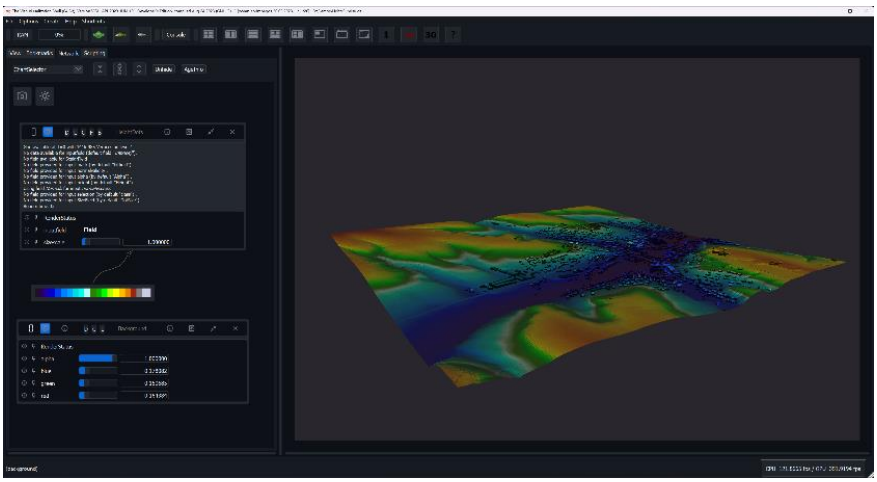


Figure 1: Vish/ the visualization shell. Visualizing PointCloud data

Content

Abstract **1**

Introduction **3**

Why is a 3D Mouse Cursor Important? 3

Existing Solutions 3

Getting Started **3**

Setting up a ray cast from the Window's Cursor Position 3

Drawing the Cursor **4**

First Model: Sphere **4**

1. Setting up OpenGL Objects 4

2. Binding the Vertex Array Object 4

3. Preparing Data Structures 5

4. Generating Sphere Vertices and Normals 5

5. Generating Indices for Triangle Strips 6
6. Setting up Vertex Buffer Object (VBO) 6
7. Setting up Element Buffer Object (EBO) 6
8. Configuring Vertex Attributes 7
9. Setting up Normal Buffer Object (NBO) 7
10. Rendering the Sphere 7
Further Optimisations 7
Facing Problems 8
Disabling OpenGL Depth Buffer 8
Cutting off the Backside 8
Floating Point Inconsistency 8
Different Settings 9
Colour and Transparency 9
Visual Enhancements 9
Using Normal Vector for Colouring 9
Drawing a Second Smaller Sphere 10
Providing Interaction Feedback 10
Second Model: Tripod 11
1. Cylinder and Cone Generation 12
2. Floating Point Precision 12
3. Depth Sorting, Back-face Culling and Colouring 13
Depth Sorting 13
Colouring 13
Rendering Process 13
Back-face Culling 14
Dynamic Scaling 14
Simple Distance-Based Scaling 14
Constrained Dynamic Scaling 15
Logarithmic Scaling 16
Alternative Approaches 16
Colour Calculation 17

Introduction

Why is a 3D Mouse Cursor Important?

Traditional 2D cursors lack depth information and spatial relationships, making them unsuitable for 3D interactions. A 3D mouse cursor enhances user engagement and makes editing tasks more intuitive by providing depth and spatial context.

Existing Solutions

Current solutions often include simple 3D arrow cursors and crosshairs. However, these typically lack features like dynamic scaling and customization options. Most don't adapt to the scene or provide real-time feedback based on cursor-camera distance or orientation.

Getting Started

Setting up a ray cast from the Window's Cursor Position

This may already be supported by your application but if not, here is a simple possible approach demonstrated which probably will need some further adjustments to fit in your program. To determine the 3D location of the cursor, we need to set up a ray cast from the 2D window coordinates. For a more detailed documentation refer to this post. Here's a basic outline:

1. Get the 2D mouse coordinates from the window system.
2. Convert these coordinates to normalized device coordinates (NDC).
3. Create a ray from the camera position through these NDC coordinates.
4. Intersect this ray with your 3D scene to determine the cursor's 3D position.

Cpp

```cpp
glm::vec3  get3DCursorPosition(int mouseX, int mouseY, const
RenderContext& Context) {
    // Convert window coordinates to NDC
    float ndcX = (2.0f * mouseX) / Context.getViewportWidth() - 1.0f;
    float ndcY = 1.0f - (2.0f * mouseY) / Context.getViewportHeight();

    // Create ray from camera through NDC point
    glm::vec4 rayClip = glm::vec4(ndcX, ndcY, -1.0, 1.0);
    glm::vec4 rayEye = glm::inverse(Context.getProjectionMatrix()) *
rayClip;

    rayEye = glm::vec4(rayEye.x, rayEye.y, -1.0, 0.0);
    glm::vec3 rayWorld =
glm::normalize(glm::vec3(glm::inverse(Context.getViewMatrix()) *
rayEye));

    // Intersect ray with your scene (simplified example)
    float distance = intersectRayWithScene(Context.getCameraPosition(),
rayWorld);
    return Context.getCameraPosition() + rayWorld * distance;
}
```

Drawing the Cursor

For our 3D cursor, we'll use a custom in-code generated model. This approach offers more flexibility and control over the cursor's appearance. Here will be demonstrated how to set up a simple sphere and an outline for a more complex tripod as cursor.

First Model: Sphere

Let's start with a sphere model. You can use whatever technique you want to draw a sphere but here's a very detailed example of a possible simple sphere generation code:

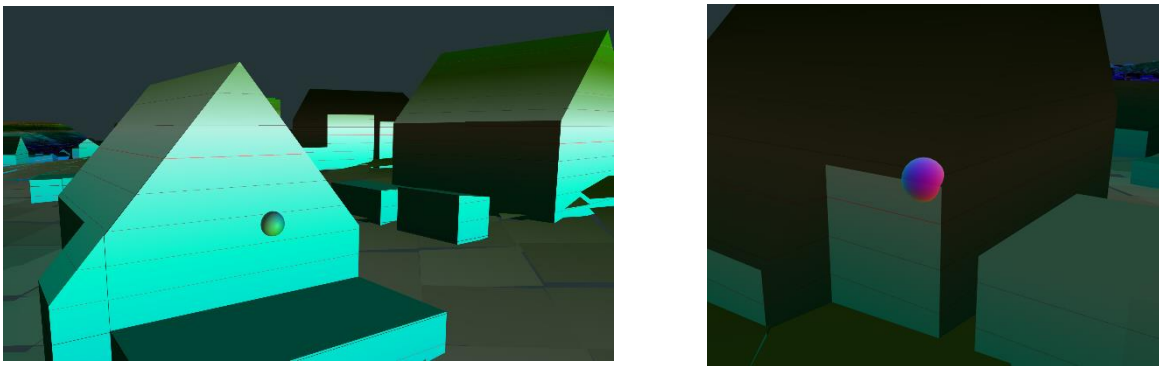


Figure 2-3: 3D Sphere Cursor in Vish/ the visualization shell. With shading(left), Without shading and slight transparency(right)

1. Setting up OpenGL Objects

First, we create the necessary OpenGL objects:

cpp

```
GLuint vao, vbo, ebo, nbo;

glGenVertexArrays(1, &vao);
glGenBuffers(1, &vbo);
glGenBuffers(1, &ebo);
glGenBuffers(1, &nbo);
```

- `vao`: Vertex Array Object - stores all of the state needed to supply vertex data
- `vbo`: Vertex Buffer Object - stores vertex data
- `ebo`: Element Buffer Object - stores indices for indexed rendering
- `nbo`: Normal Buffer Object - stores normal data

2. Binding the Vertex Array Object

cpp

```
glBindVertexArray(vao);
```

This binds the VAO, which will store all of our vertex attribute configurations and which VBOs to use.

3. Preparing Data Structures

We set up vectors to store our vertex data, indices, and normal data:

cpp

```cpp
std::vector<GLfloat> vertices;
std::vector<GLuint> indices;
std::vector<GLfloat> normalVertices;

const int latitudeBands = 30;
const int longitudeBands = 20;

vertices.resize((latitudeBands + 1) * (longitudeBands + 1) * 3);
normalVertices.resize((latitudeBands + 1) * (longitudeBands + 1) * 3);
indices.resize(latitudeBands * longitudeBands * 6);
```

The size calculations ensure we have enough space for all vertices, normals, and indices and ensures good performance. Using the `latitudeBands` and `longitudeBands` we can configure the “size” or “resolution” of our sphere.

4. Generating Sphere Vertices and Normals

This nested loop generates the vertices and normals for our sphere:

cpp

```cpp
int vertexIndex = 0;
int normalIndex = 0;

for (int lat = 0; lat <= latitudeBands; lat++) {
    float theta = lat * M_PI / latitudeBands;
    float sinTheta = sin(theta);
    float cosTheta = cos(theta);

    for (int lon = 0; lon <= longitudeBands; lon++) {
        float phi = lon * 2 * M_PI / longitudeBands;
        float sinPhi = sin(phi);
        float cosPhi = cos(phi);

        float x = cosPhi * sinTheta;
        float y = cosTheta;
        float z = sinPhi * sinTheta;

        vertices[vertexIndex++] = radius * x + LocalMouseCoordinates[0];
        vertices[vertexIndex++] = radius * y + LocalMouseCoordinates[1];
        vertices[vertexIndex++] = radius * z + LocalMouseCoordinates[2];

        normalVertices[normalIndex++] = x;
        normalVertices[normalIndex++] = y;
        normalVertices[normalIndex++] = z;
    }
}
```

This code uses spherical coordinates to generate points on a sphere. The `radius` scales the sphere, and `LocalMouseCoordinates` which we receive from our ray cast translates it to the desired position.

5. Generating Indices for Triangle Strips

This nested loop generates indices for rendering the sphere using triangle strips:

cpp

```cpp
int indexIndex = 0;

for (int lat = 0; lat < latitudeBands; lat++) {
    for (int lon = 0; lon < longitudeBands; lon++) {
        int first = lat * (longitudeBands + 1) + lon;
        int second = first + longitudeBands + 1;

        indices[indexIndex++] = first;
        indices[indexIndex++] = second;
        indices[indexIndex++] = first + 1;

        indices[indexIndex++] = second;
        indices[indexIndex++] = second + 1;
        indices[indexIndex++] = first + 1;
    }
}
```

This creates two triangles for each grid cell on the sphere's surface.

6. Setting up Vertex Buffer Object (VBO)

cpp

```cpp
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(GLfloat),
vertices.data(), GL_STATIC_DRAW);
```

This binds the VBO and copies the vertex data into it.

7. Setting up Element Buffer Object (EBO)

cpp

```cpp
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint),
indices.data(), GL_STATIC_DRAW);
```

This binds the EBO and copies the index data into it.

8. Configuring Vertex Attributes

cpp

```cpp
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
nullptr);
glEnableVertexAttribArray(0);
```

This tells OpenGL how to interpret the vertex data:

- Attribute index: 0
- Number of components per vertex attribute: 3 (x, y, z)
- Type of each component: GL_FLOAT
- Whether the data should be normalized: GL_FALSE
- Stride: 3 * sizeof(GLfloat) (distance between consecutive vertex attributes)
- Offset of the first component: nullptr (0)

9. Setting up Normal Buffer Object (NBO)

cpp

```cpp
glBindBuffer(GL_ARRAY_BUFFER, nbo);
glBufferData(GL_ARRAY_BUFFER, normalVertices.size() * sizeof(GLfloat),
normalVertices.data(), GL_STATIC_DRAW);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, nullptr);
```

This sets up the buffer for normal data, which is used for lighting calculations and later for colouring our sphere.

10. Rendering the Sphere

After setting up all the buffers, we can render the sphere with:

cpp

```cpp
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

This draws the sphere using the indices to create triangles.

Further Optimisations

The demonstrated approach whilst straight forward and simple is clearly not the best way to render our sphere. Currently we are rendering the sphere new each frame and each time calculating the spheres geometry from scratch based on the mouse position. To further enhance this, we can only generate the mesh of the sphere once and then translate it to our mouse position using the shader. This preserves not drawing the spheres geometry new each frame.

Facing Problems

We now should be able to draw a sphere at the cursors 3D position. But there still some problems we need to tackle.

Disabling OpenGL Depth Buffer

If we use the default OpenGL depth buffer, the cursor model would override its 3D position, causing it to continuously move towards the camera. To solve this, we need to disable the depth buffer for the sphere:

cpp

```cpp
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
```

Cutting off the Backside

Disabling the depth test creates a new issue: the backside of the cursor gets drawn over the front from certain view angles. We can solve this in the fragment shader by only rendering the side of the sphere which faces the camera:

Glsl

```glsl
float viewAngle = dot(NormalVector_World, normalize(relativeObserver - relative_World_Vertex.xyz));
if (viewAngle < 0) discard;
```

Floating Point Inconsistency

Another problem you might encounter in your implementation are visual glitches with the cursor if used in a very large dataset like when viewing large Point Clouds. Floating-point numbers have limited precision, especially for very large or very small values. This can lead to visual artifacts when dealing with objects far from the coordinate system's origin or when combining values of vastly different scales.

- For our 3D cursor you might notice:
- Jittering or stuttering of the cursor
- Incorrect positioning relative to other objects
- Loss of detail or precision in cursor movement

To mitigate these issues, you can implement a "floating origin" or "relative positioning" system:

- **Use a Local Coordinate System:** Instead of using world coordinates directly, maintain a local coordinate system for your cursor and nearby objects.
- **Implement a Floating Origin:** Periodically shift the origin of your coordinate system to keep objects of interest (like the cursor) near the origin.
- **Use Double Precision for Calculations:** Perform intermediate calculations using double precision, then convert to float for rendering

The need to implement one of these solutions will depend on what your use cases are and not every Solution will be suitable. These are just a few possible ways to circumvent this issue

Different Settings

Colour and Transparency

For better visibility of the cursor, we can support colour changes and transparency adjustments by letting the user chose what level of transparency and colour the sphere should have. We do this by passing the sphere colour and a separate value for transparency (which is optional) into our shader:

Glsl

```
uniform vec4 sphereColor;
uniform float transparency;

ResultColor = vec4(sphereColor.rgb, sphereColor.a * transparency);
```

For even better visibility we can make the spheres centre more transparent than the outline

Glsl

```
ResultColor.a = (1 + viewAngle) * 0.6 + 0.2;
ResultColor.a = mix(ResultColor.a, 1.0, sphereColor.a) * sphereColor.a;
```

Visual Enhancements

Using Normal Vector for Colouring

To improve spatial understanding, we can use the sphere's normal vector for colouring. This can be an optional colouring option which varies the spheres colour based of the facing direction of the spheres normal.

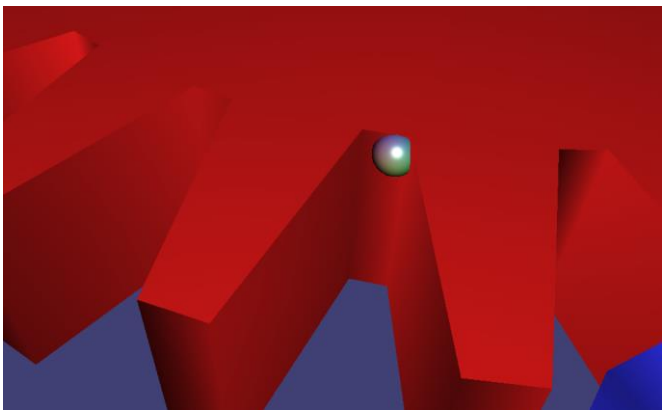


Figure 4: 3D Sphere Cursor in Vish/ the visualization shell. With simple Phong shading and normal colouring

Drawing a Second Smaller Sphere

For additional depth cues, we can draw a second, smaller sphere inside a bigger slightly transparent one:

cpp

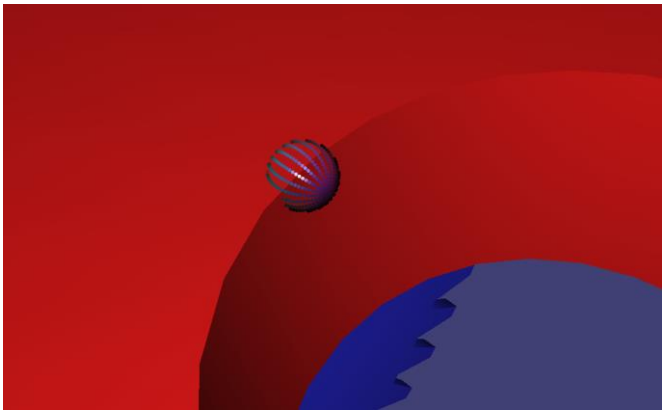
```cpp
void drawDoubleSphere(const glm::vec3 & center, double outerRadius,
double innerRadius) {
    // Draw outer sphere
    drawSphere(center, outerRadius);

    // Draw inner sphere
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    drawSphere(center, innerRadius);
    glDisable(GL_BLEND);
}
```

Providing Interaction Feedback

For better feedback you may change the spheres colour or model when the user interacts with the scene (like selecting an area or clicking to rotate the camera).

Figure 5: 3D Sphere Cursor in Vish/ the visualization shell. Illustrating user Interaction with Cursor



Cursor Position Caching vs Free Move

Currently the Cursor will work fine when navigating over meshes. However, when navigating through a point cloud or non-continuous meshes the cursor will periodically disappear when it is over the background as our collision ray check will fail. We can either cache the last position of the cursor where it has hit an object for a certain amount of time and display it there or we can only freeze the depth and move the mouse on a virtual 2D canvas over the background maintaining the last know depth and updating whenever it hits a new object. Another approach could be to always keep the same distance to the camera basically moving the cursor on a "sphere" whilst maintaining the radius of the last hit distance from the camera around if the ray cast fails.

Second Model: Tripod

The tripod cursor consists of three perpendicular axes (RGB) centred at the cursor position. Each axis is represented by a cylinder with a cone at the end, providing a clear indication of orientation in 3D space. We will not go that detailed with the Tripod but quickly outline how it could be done.

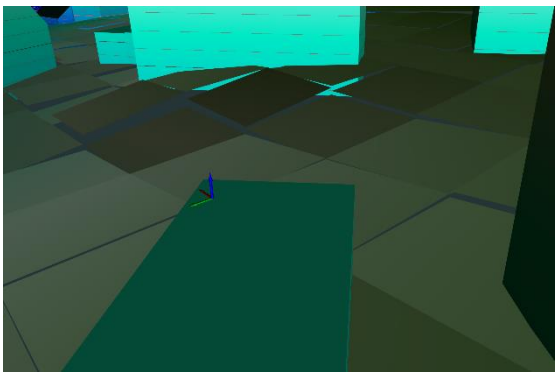


Figure 6: 3D Tripod Cursor in Vish/ the visualization shell.

1. Cylinder and Cone Generation

The `drawCylinder` function generates vertices and normals for both cylinders and cones:`cpp`

```
 void drawCylinder(double r, double length, int nTri, double close, int
face, const glm::vec3 &la) const {

   std::vector<tvector3f> P;

    std::vector<tvector3f> n;

    // Fixed size
    P.reserve(2 * nTri);
    n.reserve(2 * nTri);

    tvector3f a[3][3] = {
        {{length, 0., 0.}, {0., 1., 0.}, {0., 0., 1.}},
        {{0., length, 0.}, {0., 0., 1.}, {1., 0., 0.}},
        {{0., 0., length}, {1., 0., 0.}, {0., 1., 0.}}
    };

    // Calculate vertices
    for (int j = 0; j < nTri + 1; ++j) {
        double phi_i = static_cast<double>(j) / static_cast<double>(nTri) *
2 * M_PI;
        tvector3f N = a[face][1] * sin(phi_i) + a[face][2] * cos(phi_i);
        tvector3f Nr = N * r;
        P.push_back(la + (a[face][0] + Nr));
        P.push_back(la + (Nr * close + a[face][0] * 1.5 * (1 - close)));

        if (close == 1) {
            n.push_back(N);
            n.push_back(N);
        } else {
            tvector3f nCone = (0.5 / length * N - r / length * a[face][0]);
            n.push_back(nCone);
            n.push_back(nCone);
        }
    }
}
```

This function is versatile, creating both cylinders (`close == 1`) and cones (`close != 1`) based on input parameters.

2. Floating Point Precision

As with the sphere cursor, the tripod cursor can encounter floating point precision issues in large datasets. We should implement a local coordinate system by shifting all coordinates from the world origin like mentioned before to mitigate this.

3. Depth Sorting, Back-face Culling and Colouring

Depth Sorting

The tripod uses a sorting mechanism to render axes from back to front. As we have to disable the depth test as explained before further away axis would sometimes overdraw the nearer ones

cpp

```cpp
std::map<double, int> sortMap;

for( int i = 0; i < 3; i++){
    sortMap[normalCameraDistance[i]] = i;
}
```

This creates a map sorting the axes based on their distance from the camera. The `normalCameraDistance` is a unit vector pointing from the cursor to the camera, so its components represent how aligned each axis is with the view direction.

Colouring

Each axis of the tripod is assigned a distinct colour:

cpp

```cpp
glm::vec3 colors[3] = {{1.0,0.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0}};
```

These colours (red, green, blue) are applied to the X, Y, and Z axes respectively, providing clear visual distinction between the axes.

Rendering Process

The sorted map is then used to render the axes:

Cpp

```cpp
for(const auto &it:sortMap){
    int axisNum = it.second;
    double a = it.first;
    int isCone = (a > 0) ? 1:0;
    int isAxis = (a > 0) ? 0:1;

    // Passing which colour the axis should have into the shader
    program->setUniformValuef("axisColor", colors[axisNum], 1.0f, false);

    program->setUniformValuef("surfaceSection", -1.0f, false);
    drawCylinder(radius, lenght, 10, isCone, axisNum,
LocalMouseCoordinates);

    // ... (render the rest of the cones and cylinders)
}
```

This loop renders the axes from farthest to nearest, ensuring proper occlusion.

Back-face Culling

To prevent rendering the back sides of cylinders and cones, we let the shader know what surface is rendered and figure out if it should get displayed or not:

glsl

```
ResultColor.a = 1.0;
if(viewAngle*surfaceSection < 0) discard;
```

Here, `viewAngle` is the dot product between the surface normal and view direction. `surfaceSection` is set to either 1.0 or -1.0 to determine which side of the surface to render. This ensures only the front-facing parts of each axis are visible.

Alternative Approaches

Instead of drawing a tripod displaying only the three positive axes you may also display a 3D crosshair. Using what's been described already we can simply rotate our tripod and place it next to the already existing one which results in a 3D crosshair. We may also combine the crosshair with our generated sphere by offsetting the axes in their pointing directions and rendering the sphere in the center.

Dynamic Scaling

In 3D environments, maintaining an appropriate cursor size is crucial for usability. A cursor that's too small when far away becomes hard to see, while one that's too large when close can obstruct the view. All examples provide the user with the option to change the base scaling size of the cursor for further customization.

Simple Distance-Based Scaling

A basic approach is to scale the cursor linearly based on its distance from the camera:

cpp

```
float sphereRadius = fixedSphereRad * glm::distance(cursorPosition,
cameraPosition)
```

Pros:

- Easy to implement
- Ensures cursor is always visible

Cons:

- Loss of depth perception as the cursor appears the same size regardless of distance

Constrained Dynamic Scaling

This method creates a min-max threshold for cursor scaling based on its relative screen size. So, the cursor always has a min/max size on the screen. If it would exceed this threshold, it maintains the same size on screen (scaling like in the linear distance-based approach). This allows better depth perception as the cursor get smaller moving away and bigger when moving towards the camera whilst never getting too small or too big. In this example we are saving the oldSphereRadius between the function calls so if the radius is in the boundaries, it stays the same i.e. retaining depth perception:

cpp

```
    double distanceFactor = std::sqrt(glm::dot(cameraDistance,
cameraDistance));

    double defaultScreenSize = std::pow(fixedSphereRad, 2) *
distanceFactor;
    double minScreenSize = std::pow(fixedSphereRad - minDiff, 2) *
distanceFactor;
    double maxScreenSize = std::pow(fixedSphereRad + maxDiff, 2) *
distanceFactor;

    float sphereRadius = clamp(oldSphereRadius, minScreenSize,
maxScreenSize);
oldSphereRadius = sphereRadius;
```

Pros:

- Maintains depth perception within a certain range
- Prevents the cursor from becoming too small or too large
- Allows for customization (minDiff, maxDiff)

Cons:

- More complex to implement
- May have weird size changes at threshold boundaries

Logarithmic Scaling

Another approach is to use simple logarithmic scaling:

cpp

```cpp
float sphereRadius = fixdSphereRad * (1 +
log(glm::distance(cursorPosition, cameraPosition)));
```

Pros:

- Smooth scaling across all distances
- Maintains some depth perception

Cons:

- May still become too small or large at extreme distances
- Harder to fine-tune for specific use cases

Alternative Approaches

The “On Geometry” Drawing Mode is an alternative approach to rendering 3D cursors directly on geometry, rather than as a separate 3D object. This technique can be particularly useful in certain scenarios where you want the cursor to interact more closely with the underlying geometry. This implementation is actually pretty simple and can be done mostly inside the fragment shader. We only need to pass the cursors position inside the shader as a uniform variable and then draw our circle.

Glsl

```glsl
vec3 cameraDistance = Observer - circleCenter;

//scaling linear to the camera distance
float circleRad = defaultCircleRad * defaultCircleRad *
(sqrt(dot(cameraDistance,cameraDistance)));
vec3 distanceVec = WorldFragmentCoordinates - circleCenter;
vec3 distanceVec2 = distanceVec * (1 + circleBorderThickness);
float centerDistance = dot(distanceVec, distanceVec) - circleRad *
circleRad;
float centerDistance2 = dot(distanceVec2, distanceVec2) - circleRad *
circleRad;

// ... (color calculations)

FragColor.rgb = mix(FragColor.rgb, realColor, step(centerDistance, 0.0)
* step(-centerDistance2, 0.0));
```

This shader code calculates the distance of each fragment from the cursor centre and uses this to determine whether to draw the outline or not. To further enhance the cursor, we let the user pick a default `defaultCircleRad` and then scale it with the distance of the camera to the cursor. Also letting the user choose the `circleBorderThickness` is possible in this example.

Colour Calculation

To further improve visibility, we can calculate the `complementaryColor` of the underlying mesh and use this for our cursor. This keeps the cursor visible on every texture. The downside being that on complex textures the colour could get to colourful and hard to see. We can circumvent this by introducing a threshold in which the colours are getting combined to an average. If a colour exceeds the threshold it creates a new average with similar colors. This way the cursor will only have a limited number of different colours one at a time. Here is a simple approach of calculating the complementary colour and blending it with a user selected colour:

Glsl

```
vec3 complementaryColor = vec3(1,1,1) - FragColor.rgb;
vec3 realColor = mix(circleColor.rgb, complementaryColor, circleColor.a);
```



Figure 7-9: Cursor in Vish/ the visualization shell with "On Geometry" drawing mode on a Point Cloud