UNIVERSITY OF MINNESOTA
Driven to Discover®

Ray Casting

CSCI 4611: Programming Interactive Computer Graphics and Games

Evan Suma Rosenberg | CSCI 4611 | Fall 2022

Review: The Synthetic Camera

The **synthetic camera** is the programmer's model to specify 3D view projection parameters.

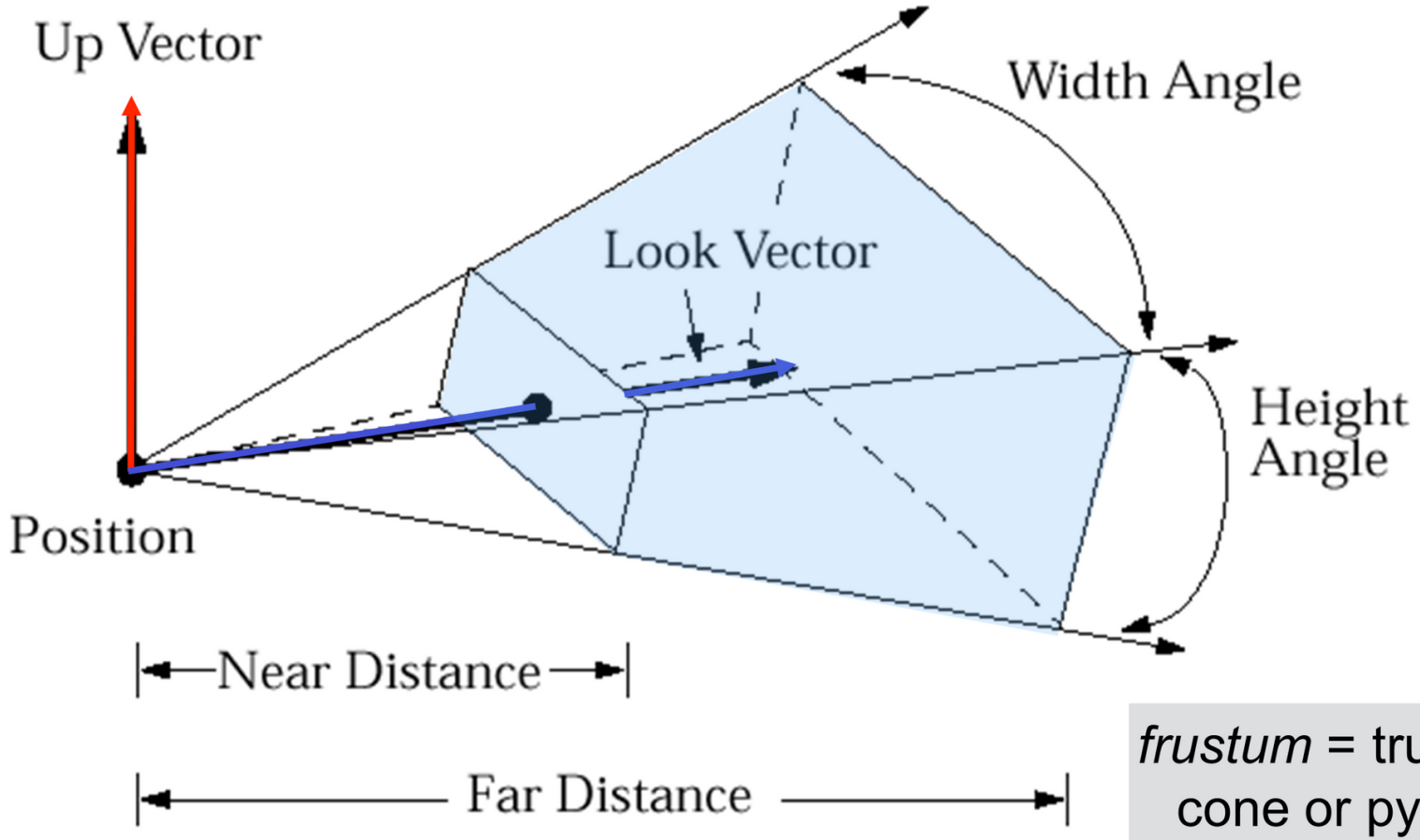
Each graphics package has its own but they are all (nearly) equivalent.

General synthetic camera:

- position and orientation of camera
- field of view
- aspect ratio
- near and far clipping distances

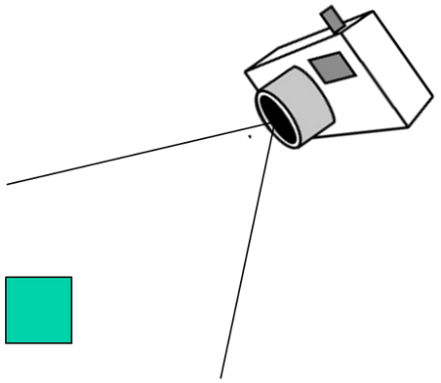
  perspective or parallel projection

Perspective View Volume

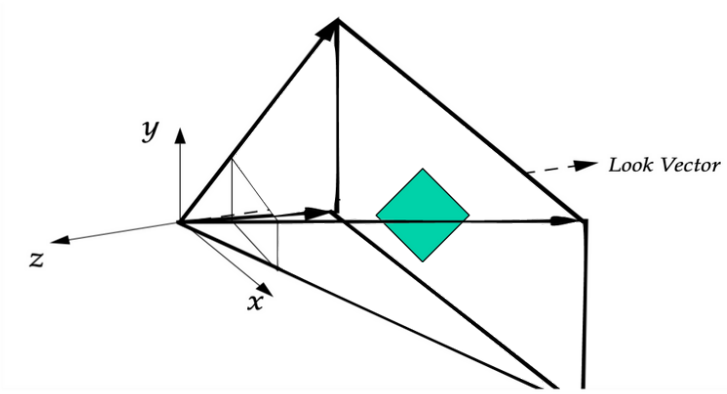


frustum = truncated cone or pyramid

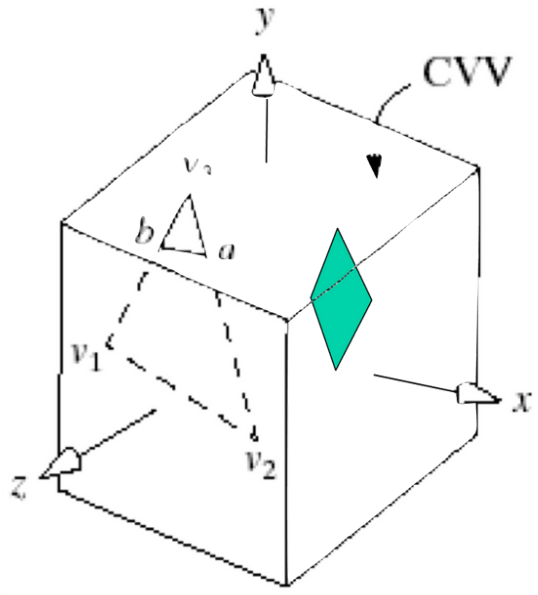
world space



eye space



canonical view volume



view matrix

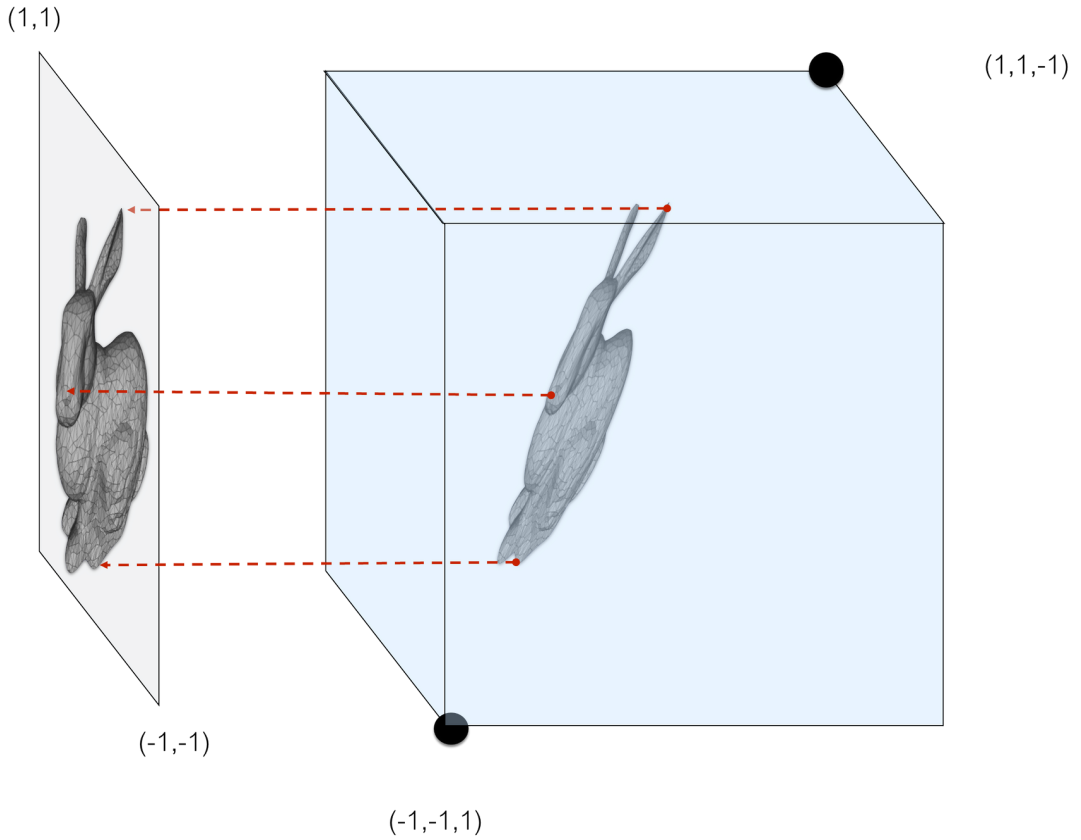
proj matrix

These two matrices implement all the parameters of our synthetic camera model!

Canonical View Volume to Screen Coordinates

The convert to 2D screen coordinates, we just "drop" the Z coordinate.

However, we need to make sure to preserve *relative* depth ordering so closer pixels are drawn on top of ones that are further away.



The “Z Buffer” and “Z Test”

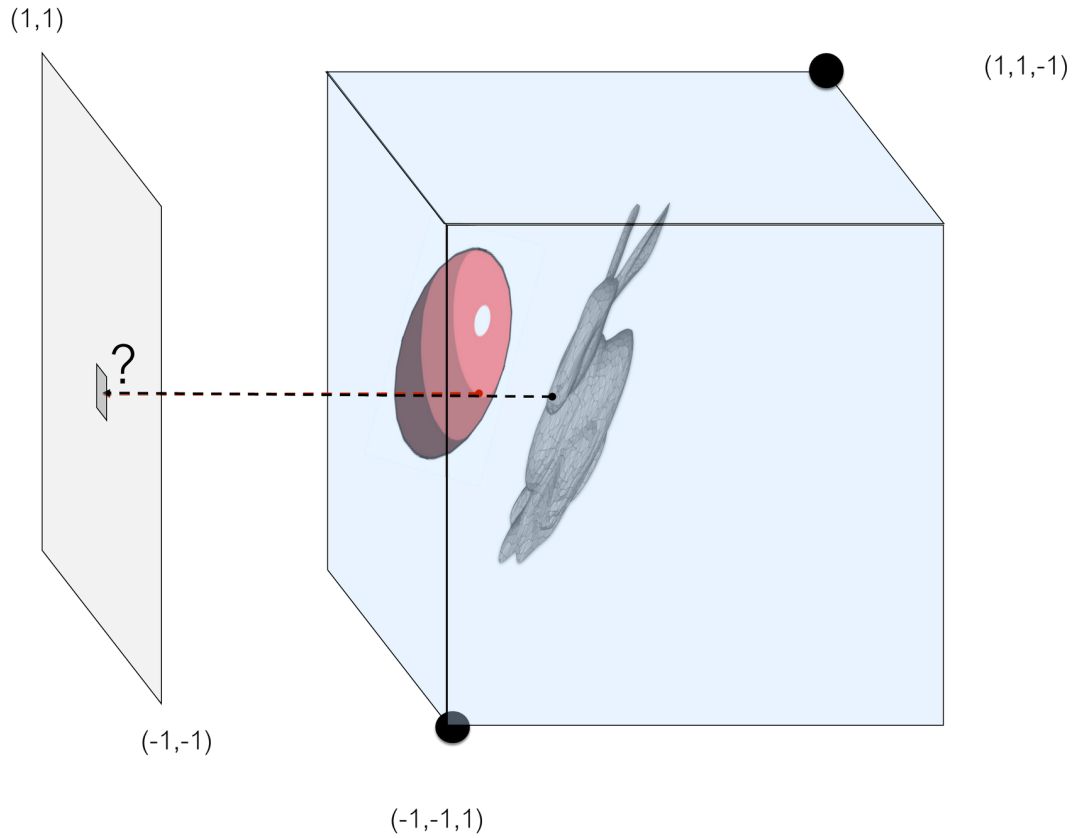
Every time we write (*r,g,b*) to the color buffer, we also write **pseudodepth** to a **Z buffer**.

Before rendering each fragment, the rasterizer has already figured out which pixel on the screen the fragment will be projected to.

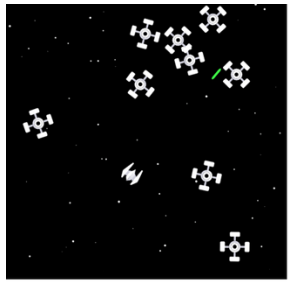
So, we look up the current pseudodepth for that pixel in the Z-Buffer and compare it to the pseudodepth for the new fragment.

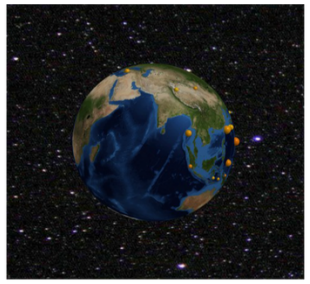
If the new fragment is closer, it passes the **Z Test**, and we run the fragment shader and write to the color buffer and Z buffer.

If the Z Test fails, we know there is some other object closer to the camera that blocks the view of this fragment, so we discard it.



Semester Recap







1. First interactive game with 2D graphics
2. Points, vectors, essential computer graphics math, simple physical simulation.
3. Intro to 3D modeling, tessellating complex shapes with triangles, morphing, storing data on the GPU.
4. Hierarchical transformation matrices, character animation from mocap data.
5. Writing your first shaders, the continuum from per-pixel physics-based to art-based shading.
6. Mouse-based 3D interfaces and interactive free-form 3D modeling.

User Interaction with 3D Graphics

- In Space Minesweeper, we used a mouse to indicate 2D direction and speed of the ship.
- In Car Soccer, we used the keyboard to drive a car around.
- In Earthquake Visualization and Artistic Rendering, we used a couple of methods for moving the virtual camera around in a 3D scene using the mouse.
- We haven't yet used the mouse to ***directly*** select or manipulate 3D objects in the scene.

Let's Start with 3D Selection

Let's say we click the mouse on a 3D object in our scene.

We know:

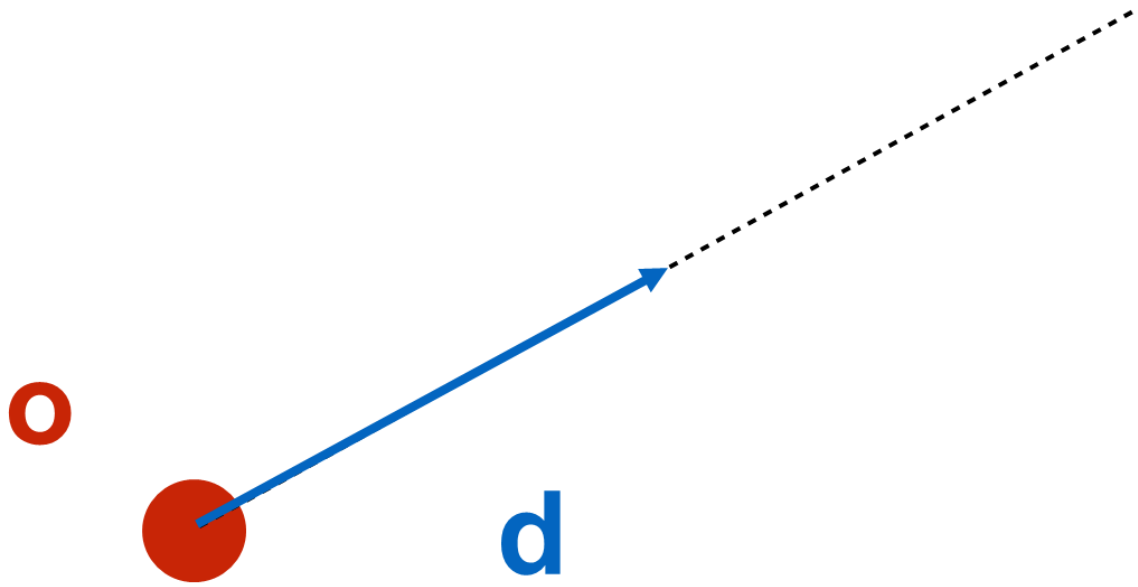
- Mouse 2D position within the window when it was clicked.
- Details of the 3D scene (camera placement, model matrix for each object, shape of each object).

How do we figure out which object we clicked?

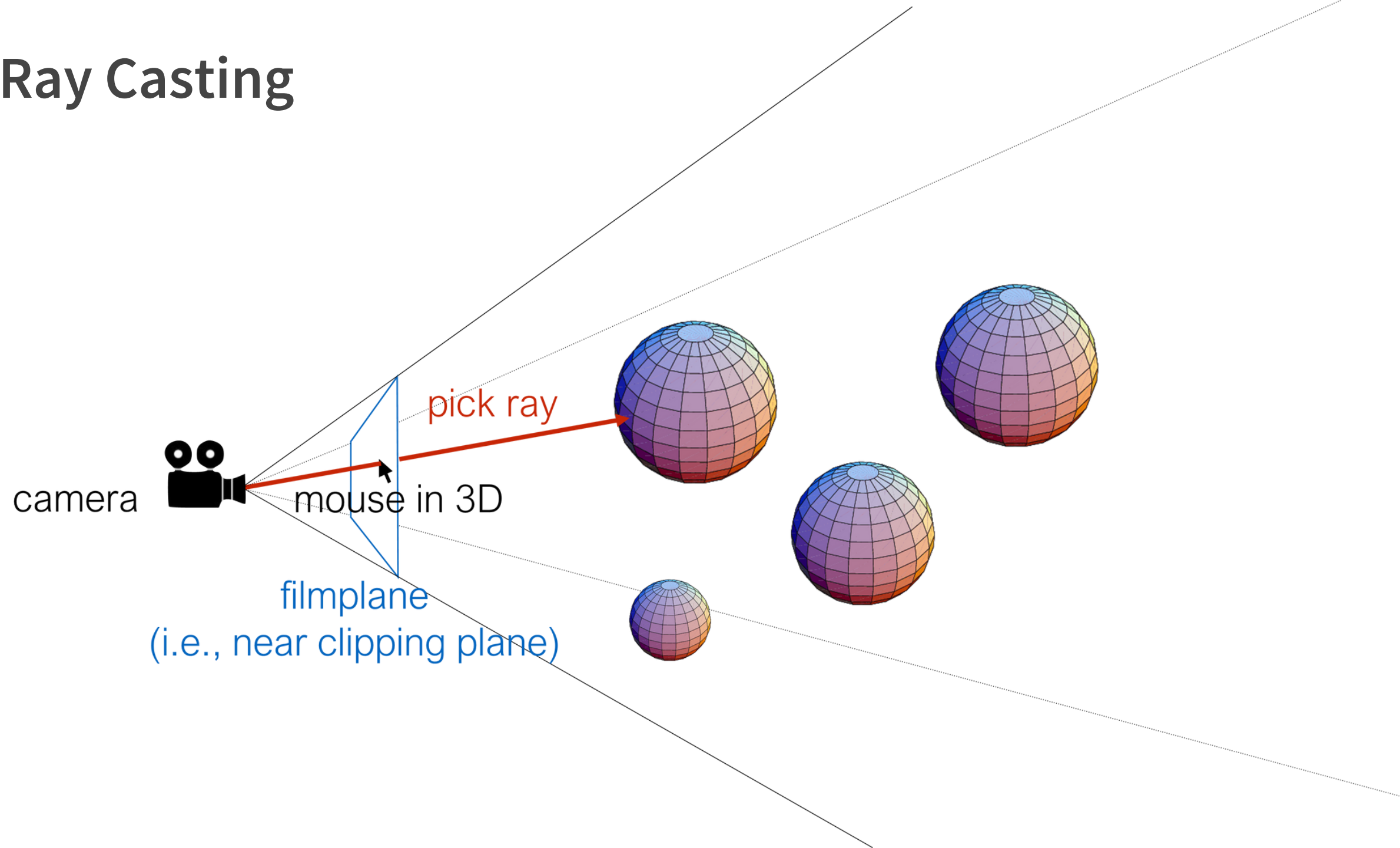
Add Rays to your Essential Graphics Math Toolbox

A **ray** is defined by its origin (a point) and a direction (a vector).

Any point on this ray is $\mathbf{o} + t\mathbf{d}$ for some scalar $t \geq 0$.



Ray Casting


pick ray
camera
mouse in 3D
filmplane
(i.e., near clipping plane)

3D Selection with a Pick Ray (Single Mesh)

```
onMouseDown(event: MouseEvent): void
{
    // Normalized Device Coordinates go from -1 to +1 in x and y.
    const deviceCoords = this.getDeviceCoordinates(event.x, event.y);

    const ray = new gfx.Ray();
    ray.setPickRay(deviceCoords, this.camera);

    const intersection = ray.intersectsMesh( a_single_3D_mesh_in_the_scene );
    if(intersection)
    {
        // If the ray intersects the mesh, then the method returns a Vector3.
        // If there is no intersection, then it returns null.
    }
}
```

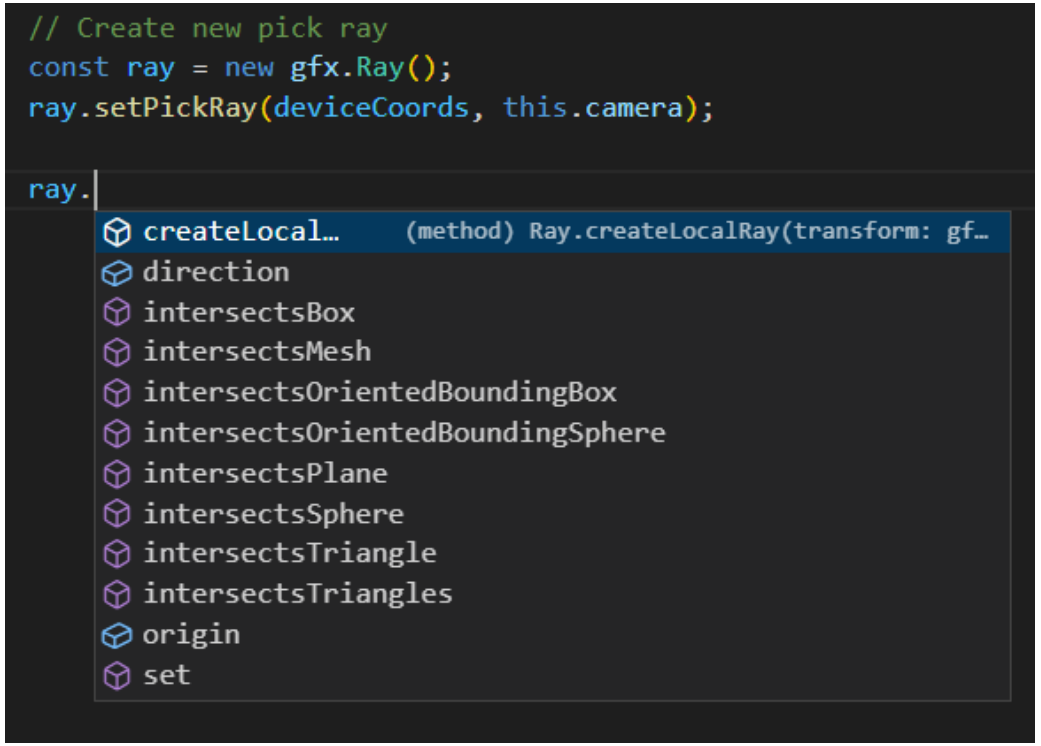
Ray Intersection Tests

We must loop through every triangle in the mesh and compute a ray-triangle intersection.

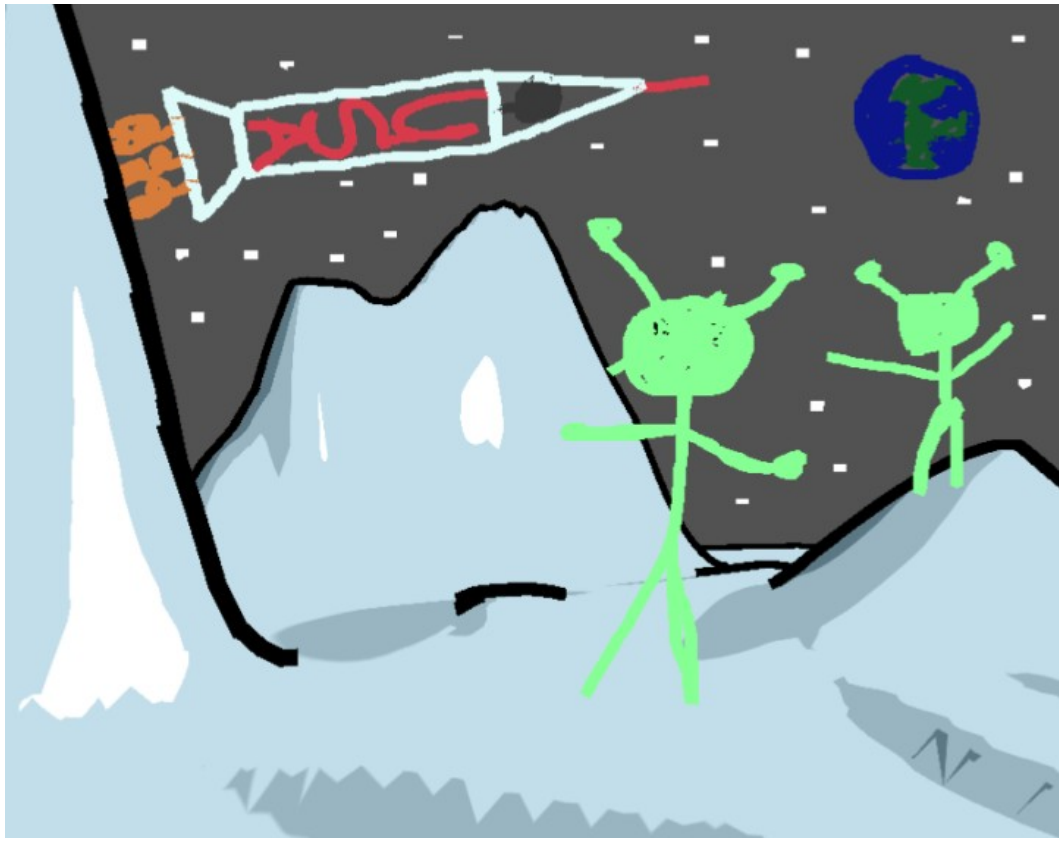
GopherGfx implements the Möller–Trumbore intersection algorithm.

We should therefore use the intersectsMesh() method only when absolutely necessary.

Perform intersection tests using simplified proxy geometry when possible.



Assignment 6: A World Made of Drawings



Inspired by Harold: A World Made of Drawings by Cohen et al. [ACM NPAR 2000]

https://mediaspace.umn.edu/media/t/1_gtj35asj

Three Key Features in our Version

1. Drawing on the sky
2. Editing the ground
3. Creating and adding to billboards

Mouse-Based User Interface

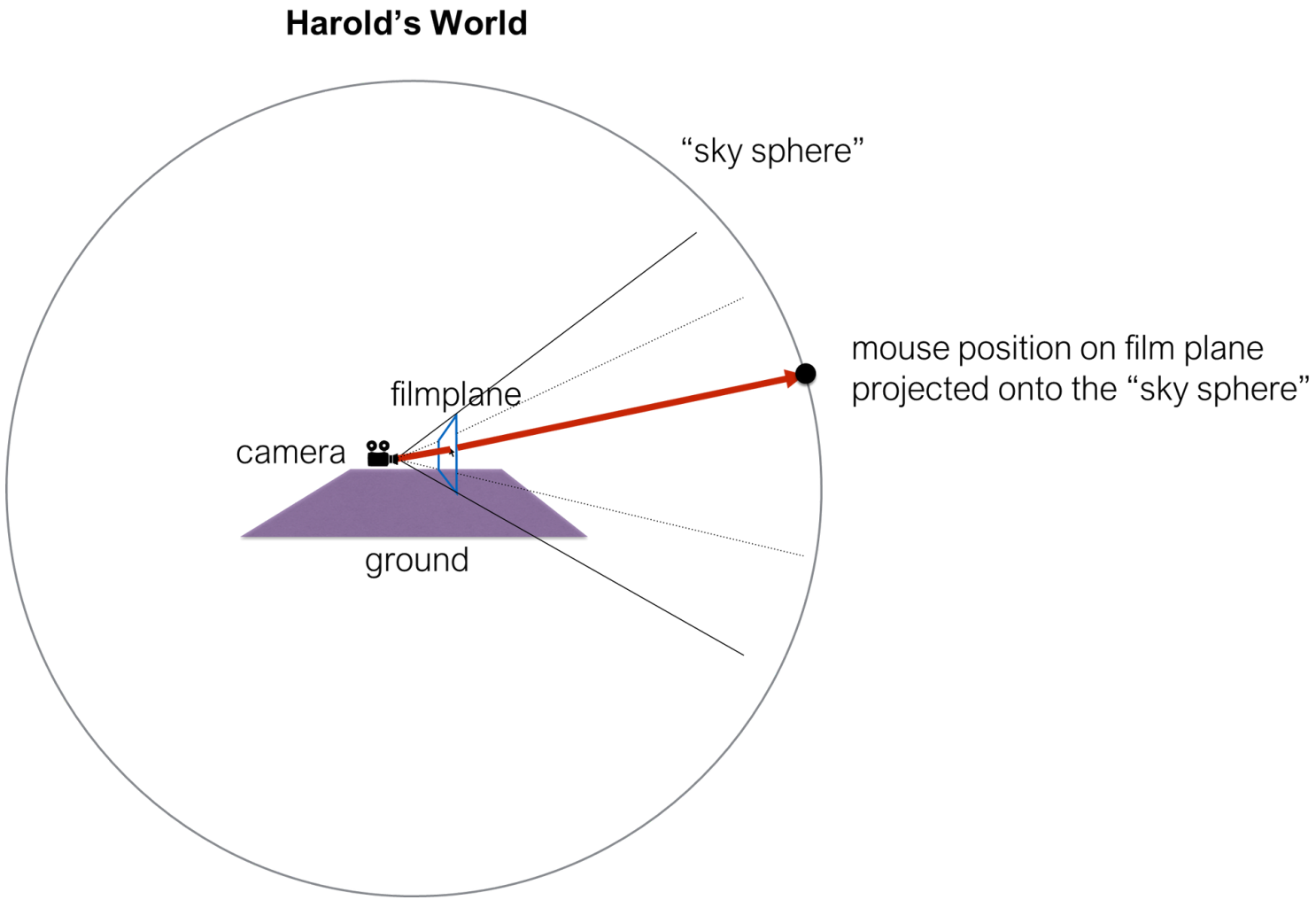
We don't need any mode buttons in this user interface because we can **infer** the user's intent from the **context**.

Stroke Made by Mouse	3D Modeling Operation
Starts in the sky	Add a new stroke to the sky
Starts AND ends on the ground	Edit the ground mesh to create hills and valleys
Starts on the ground and ends in the sky	Create a new billboard
Starts on an existing billboard	Add the stroke to the existing billboard

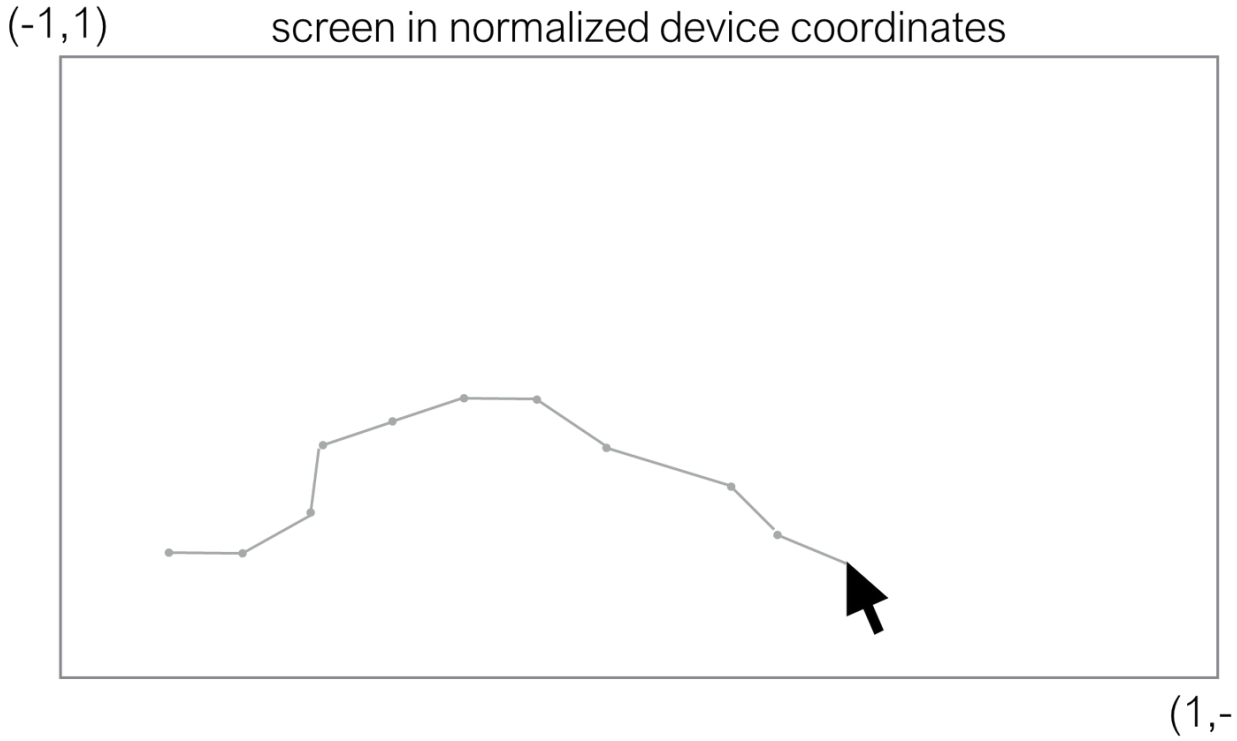
Live Demo

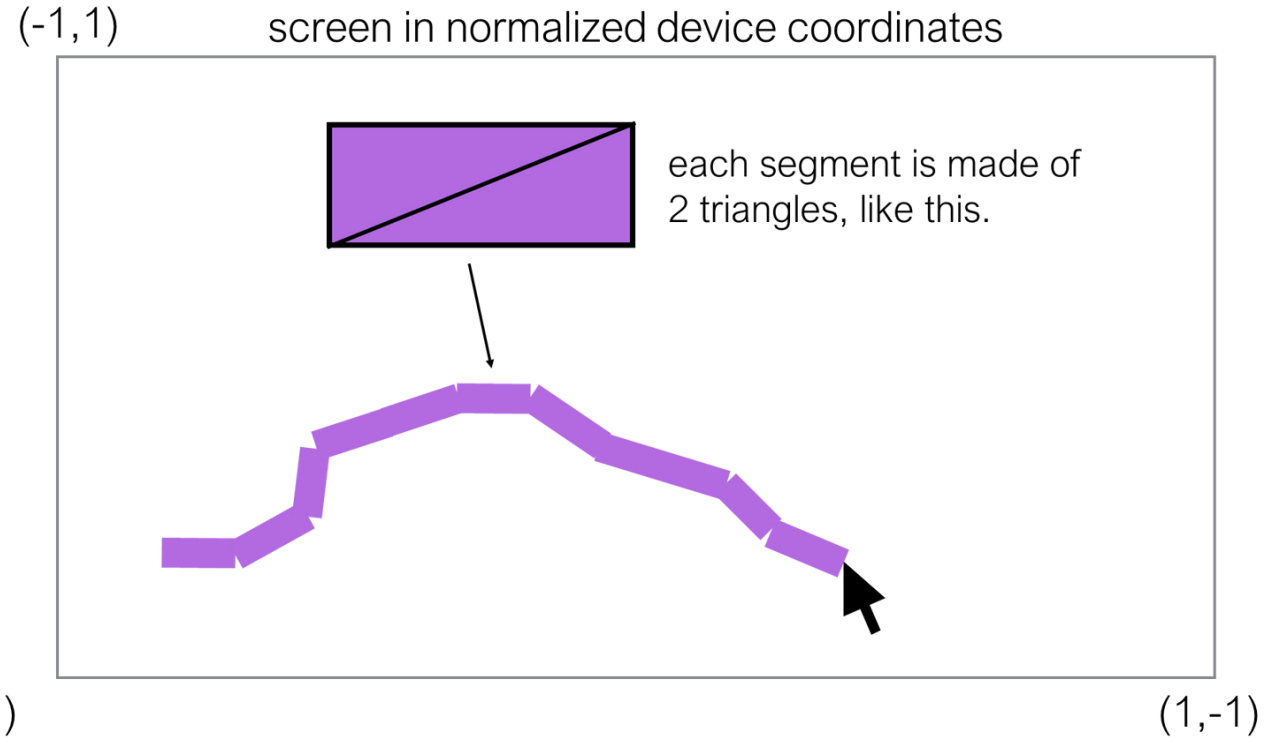
https://csci-4611-fall-2022.github.io/Builds/Assignment-6/

Feature #1: Drawing on the Sky



Drawing a 2D “stroke” with the mouse on the screen



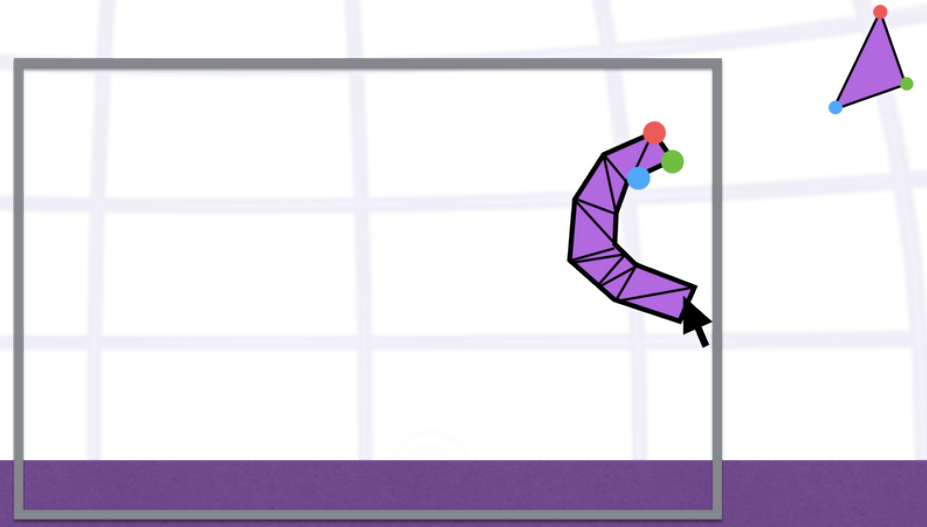


2D mouse movement on the screen generates a series of OnMouseMove() events. If we collect these Vector2 objects in an array of mouse positions, we can create a stroke, connecting them together into a polyline to approximate a curved path.

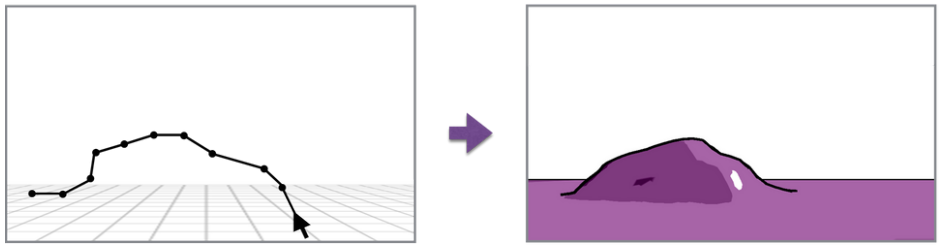
We can't draw the polyline directly, we need to convert it into triangles first so that we get a nice thick purple line to show up on the screen. So, we create a simple mesh using the stroke as the centerline. This is a 3D mesh with the XY normalized device coordinates and a constant Z value of -0.999.

Projecting to the Sky

- Create a *pick ray* for each vertex in the 2D mesh.
- Shoot that ray out into 3D space until it intersects the inside of the sky (typically represented as a box or sphere).
- Move the vertex to the 3D point of intersection.
- If you do this correctly, you should see no change when your program converts from a 2D mesh to a 3D mesh because the vertices should still project to the exact same 2D location on the screen. It won't look like the stroke has moved at all until you start walking around!



Feature #2: Editing the Ground



Terrain-editing strokes must start and end on the ground. Call the starting and ending points S and E… [T]hese two points, together with the **y**-vector, determine a plane in R^3, that we call the *projection plane*. The points of the terrain-editing stroke are projected onto this plane (this projection, which is a curve in R^3, is called the *silhouette curve*); the shadow of the resulting curve (as cast by a sun directly overhead) is a path on the ground (we call this the *shadow*). Points near the shadow have their elevation altered by a rule: each point P near the shadow computes its new height (y-value), P'_y, as a convex combination[1]

$$P'_y = \begin{cases} (1 - w(d)) \cdot P_y + w(d) \cdot h & \text{if } h \neq 0 \\ P_y & \text{if } h = 0 \end{cases}$$

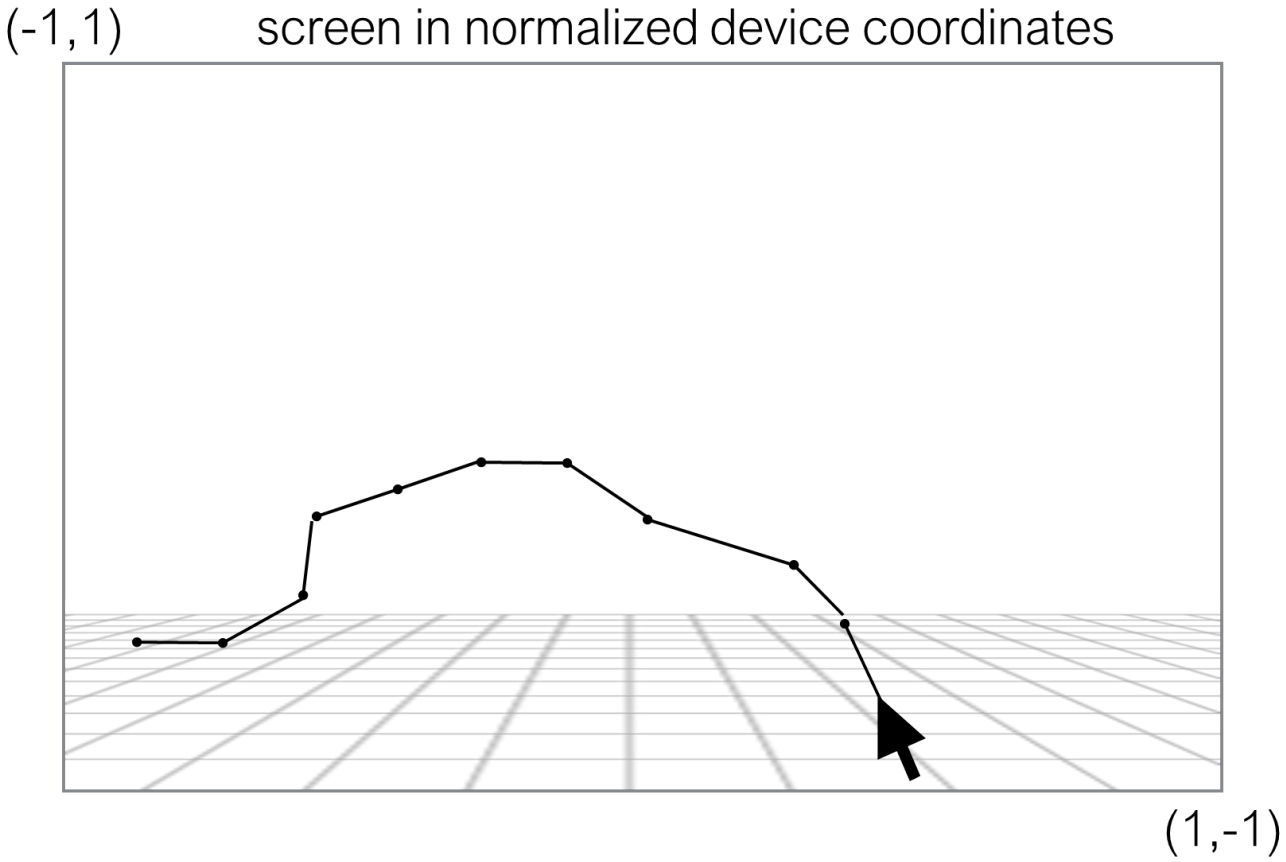
where d is the distance from P to the projection plane, h is the y-value of the silhouette curve over the nearest point on the projection plane to P, and $w(d)$ is a weighting function given by

$$w(d) = \max\left(0, 1 - \left(\frac{d}{5}\right)^2\right).$$

This gives a parabolic cross-section of width 10 for a curve drawn over level terrain. Other choices for w would yield hills with different shapes that might be more intuitive, but this particular choice gives reasonable results in most cases.

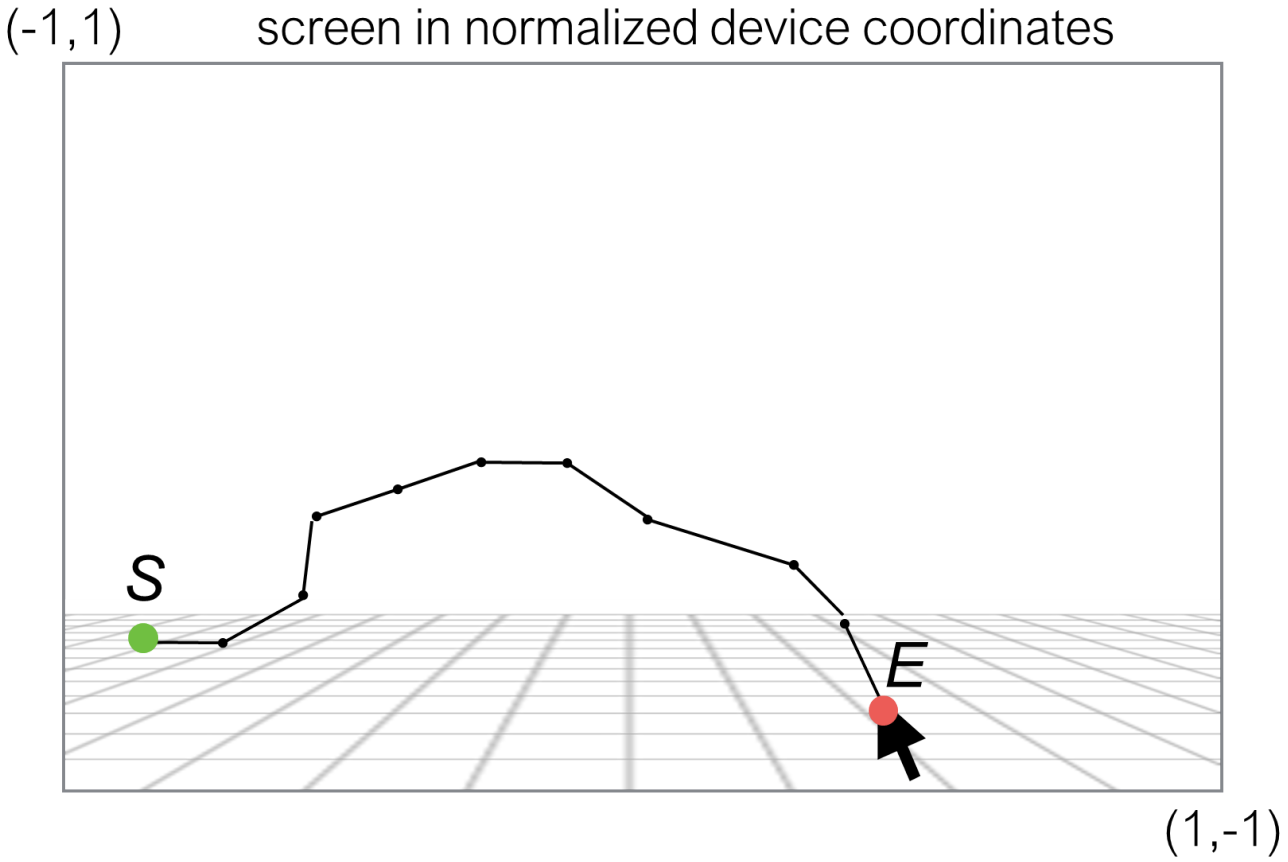
Note that if the silhouette curve bends back on itself (i.e. it defines a silhouette that cannot be modeled using a heightfield), then the variation of height along the shadow will be discontinuous. The resulting terrain then may have unexpected features.

Terrain Editing Algorithm Step 1



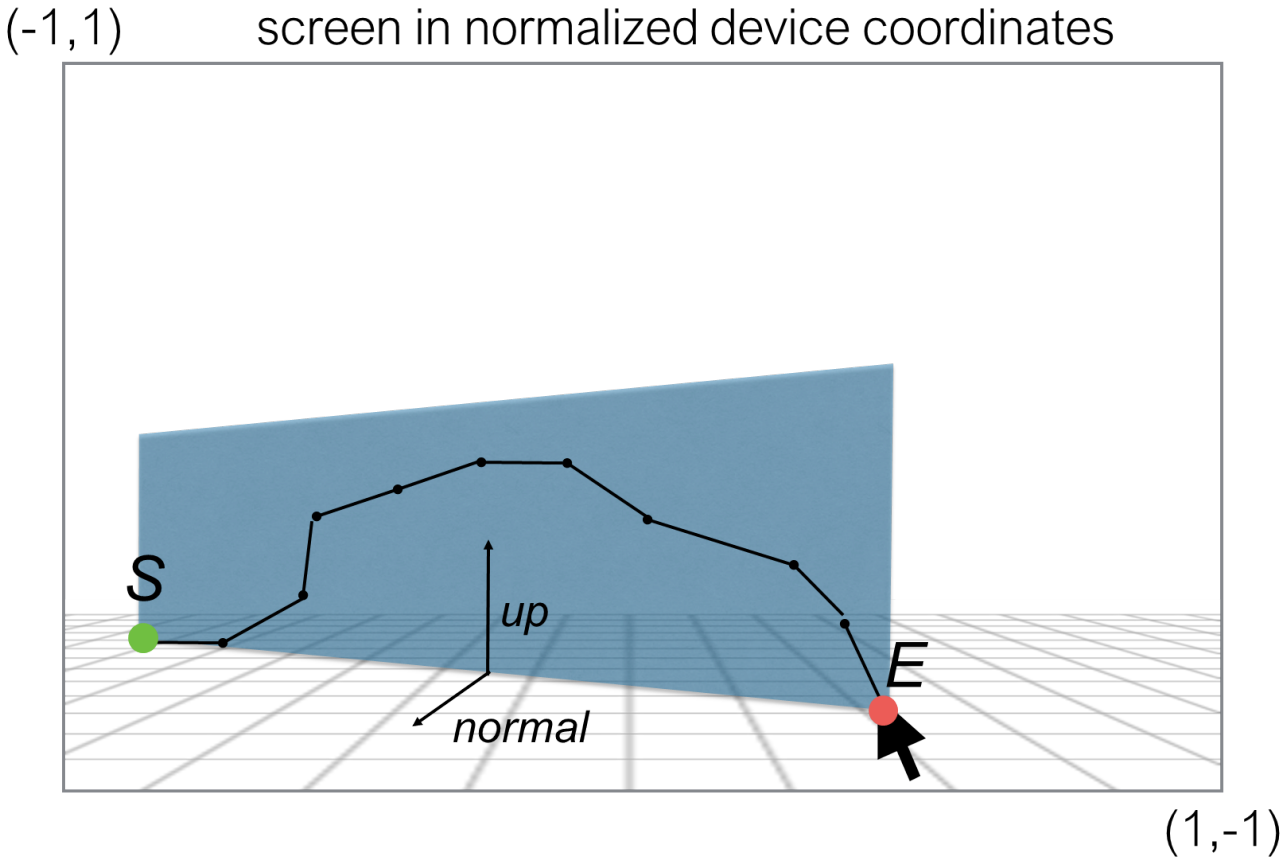
- Define a *projection plane* in 3D using the first and last point of the stroke.

Terrain Editing Algorithm Step 1



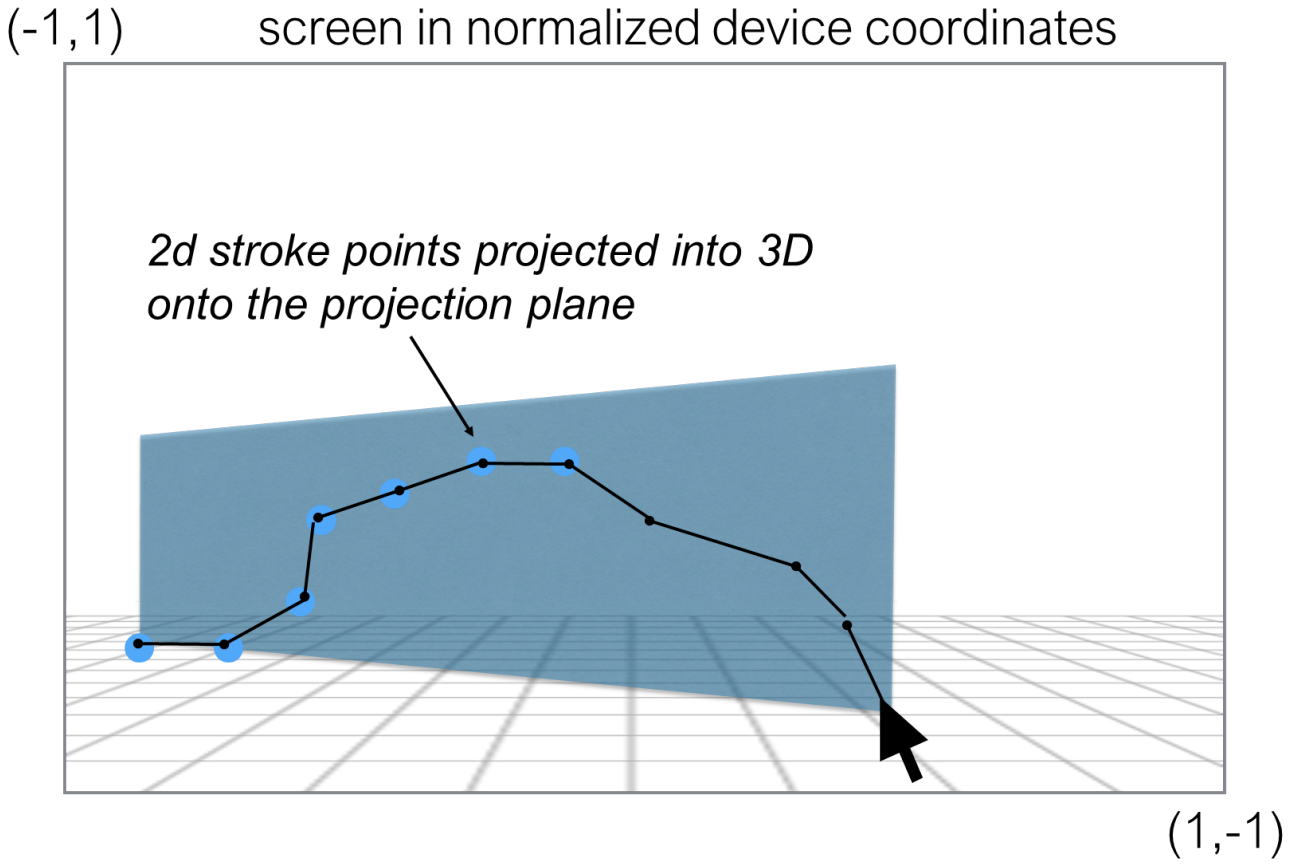
- Define a *projection plane* in 3D using the first and last point of the stroke.
- ***S*** = start of stroke, projected onto the ground in 3D
- ***E*** = end of stroke, projected onto the ground in 3D

Terrain Editing Algorithm Step 1



- Define a *projection plane* in 3D using the first and last point of the stroke.
- ***S*** = start of stroke, projected onto the ground in 3D
- ***E*** = end of stroke, projected onto the ground in 3D
- up = ???
- normal = ???

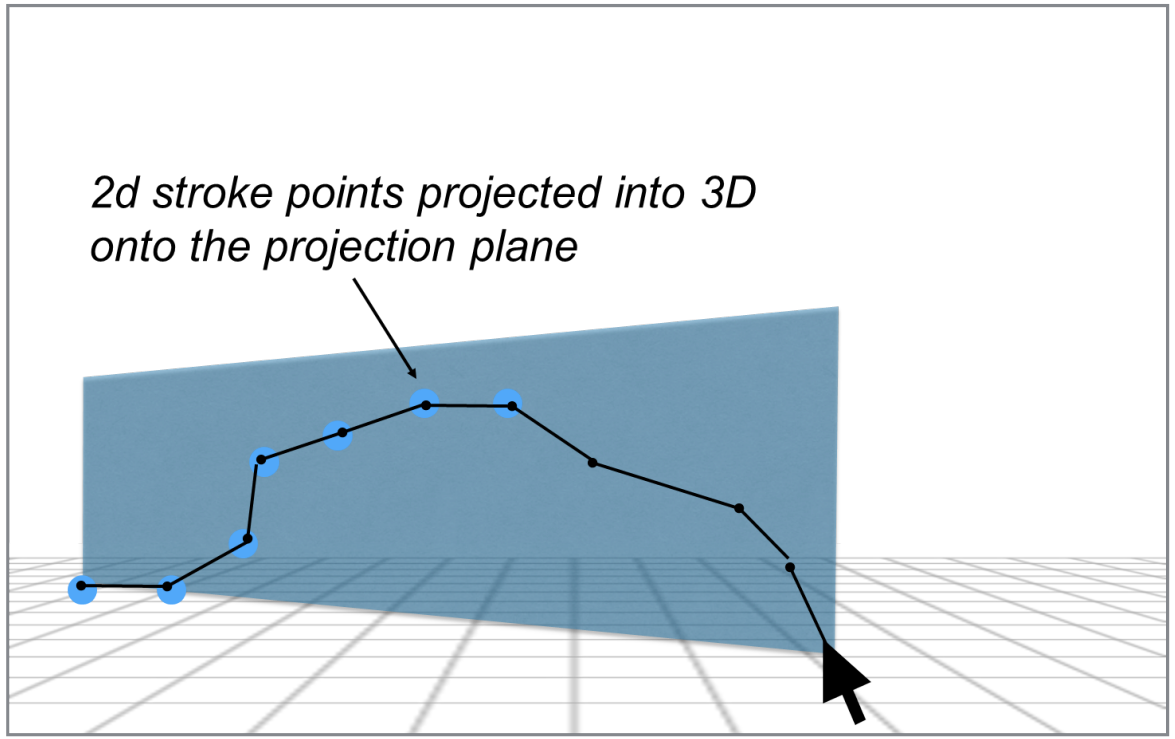
Terrain Editing Algorithm Step 2



- Project the 2D stroke onto the projection plane to create the silhouette curve.
- How do we do this projection?

Terrain Editing Algorithm Step 2

(-1,1) screen in normalized device coordinates



(1,-1)

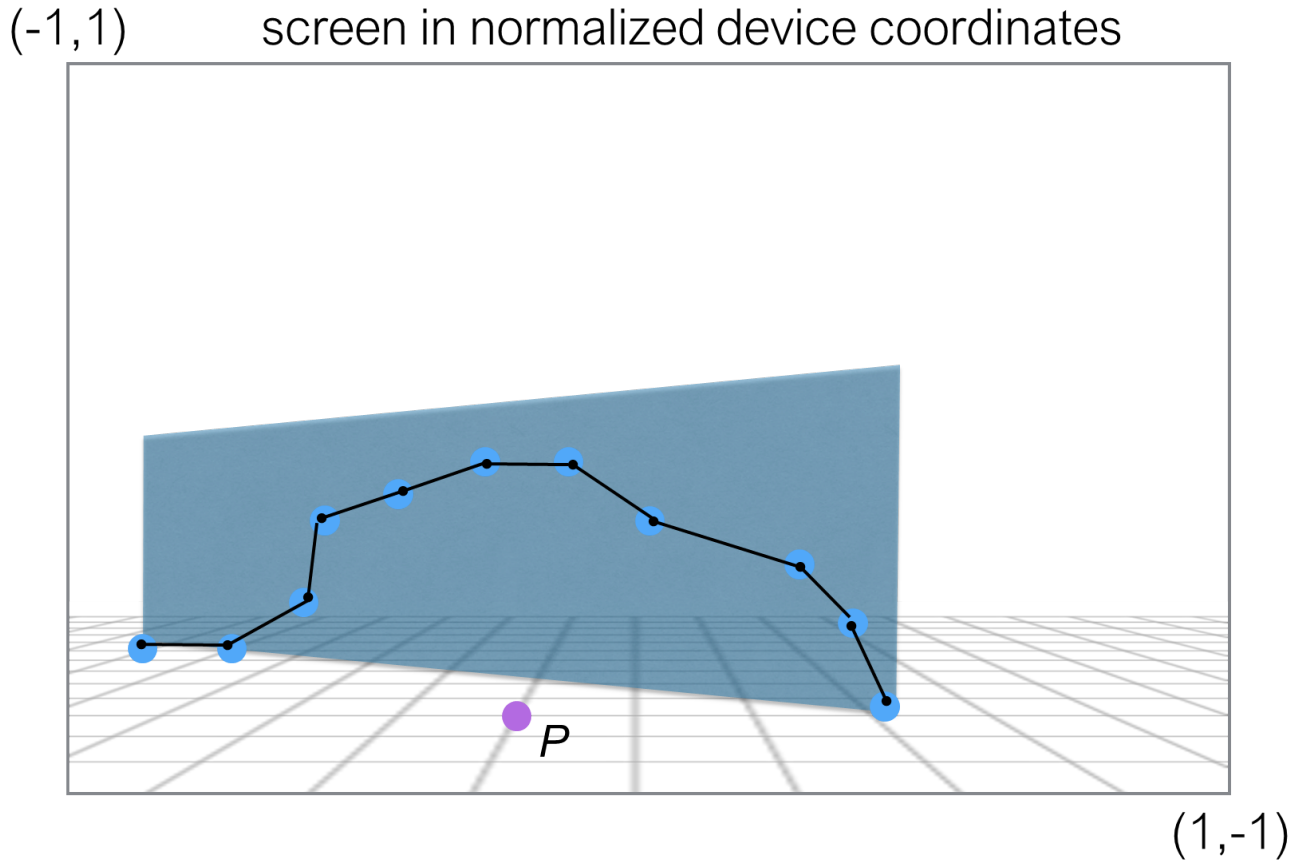
- Project the 2D stroke onto the projection plane to create the silhouette curve.
- How do we do this projection?

```
// Create new pick ray
const ray = new gfx.Ray();
ray.setPickRay(deviceCoords, this.camera);

ray.
```

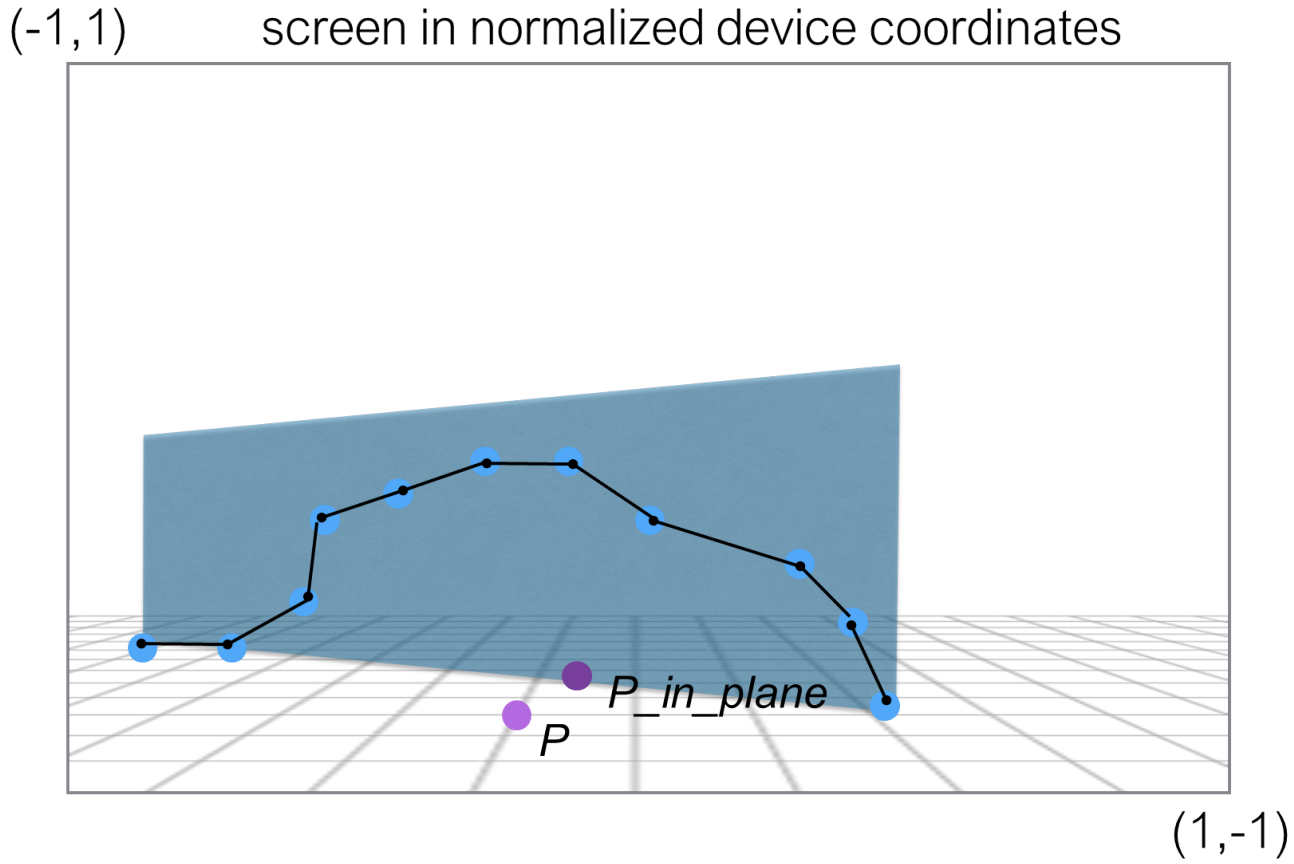
createLocal… (method) Ray.createLocalRay(transform: gf…
direction
intersectsBox
intersectsMesh
intersectsOrientedBoundingBox
intersectsOrientedBoundingSphere
intersectsPlane
intersectsSphere
intersectsTriangle
intersectsTriangles
origin
set

Terrain Editing Algorithm Step 3



For each vertex **P** in the ground mesh:

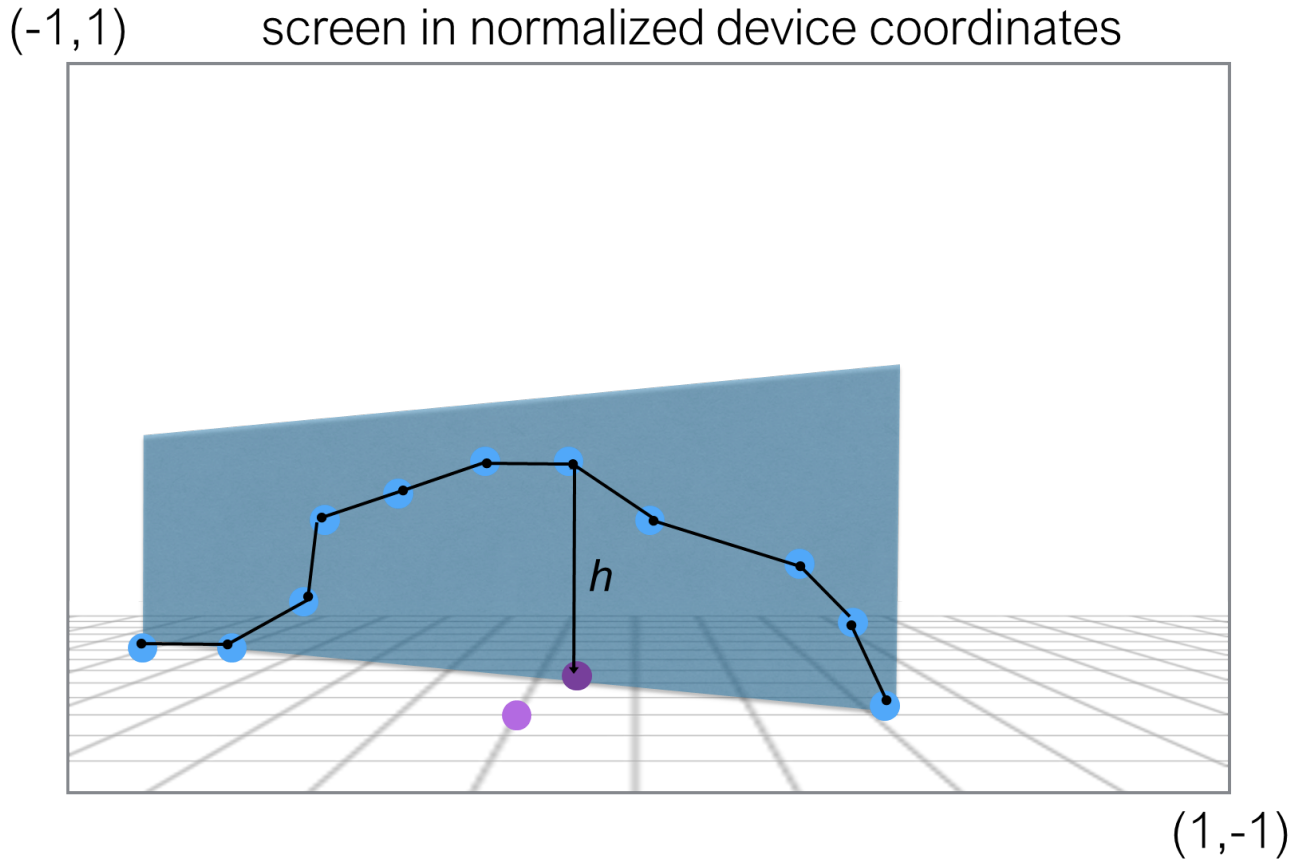
Terrain Editing Algorithm Step 3



For each vertex **P** in the ground mesh:

- Find the **closest point** to **P** that lies within the projection plane.
- *Hint: gfx.Plane has a project() method that will compute a scalar projection of a point onto the plane.*

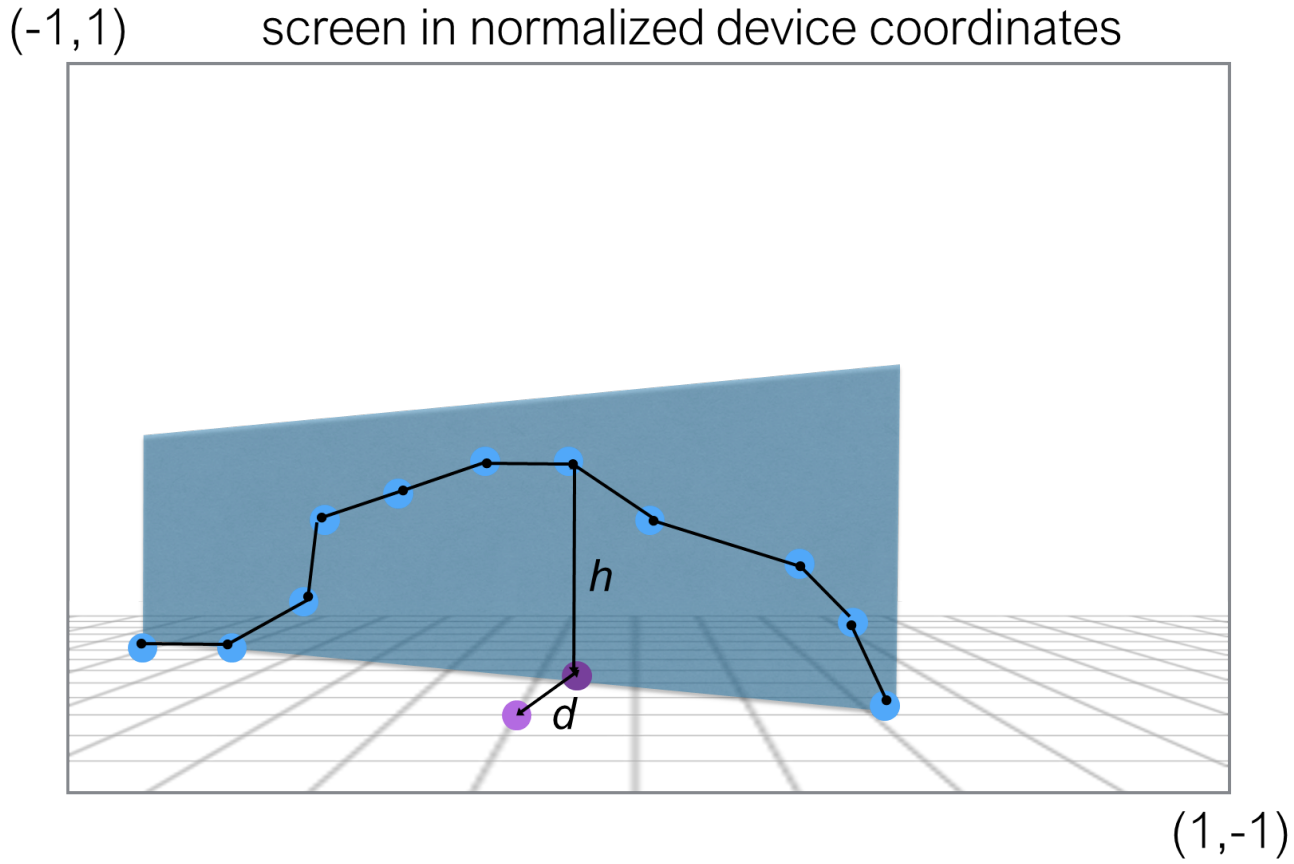
Terrain Editing Algorithm Step 3



For each vertex **P** in the ground mesh:

- Find the **closest point** to **P** that lies within the projection plane.
- Find **h**, the height of the silhouette curve relative to **P_in_plane**.

Terrain Editing Algorithm Step 3



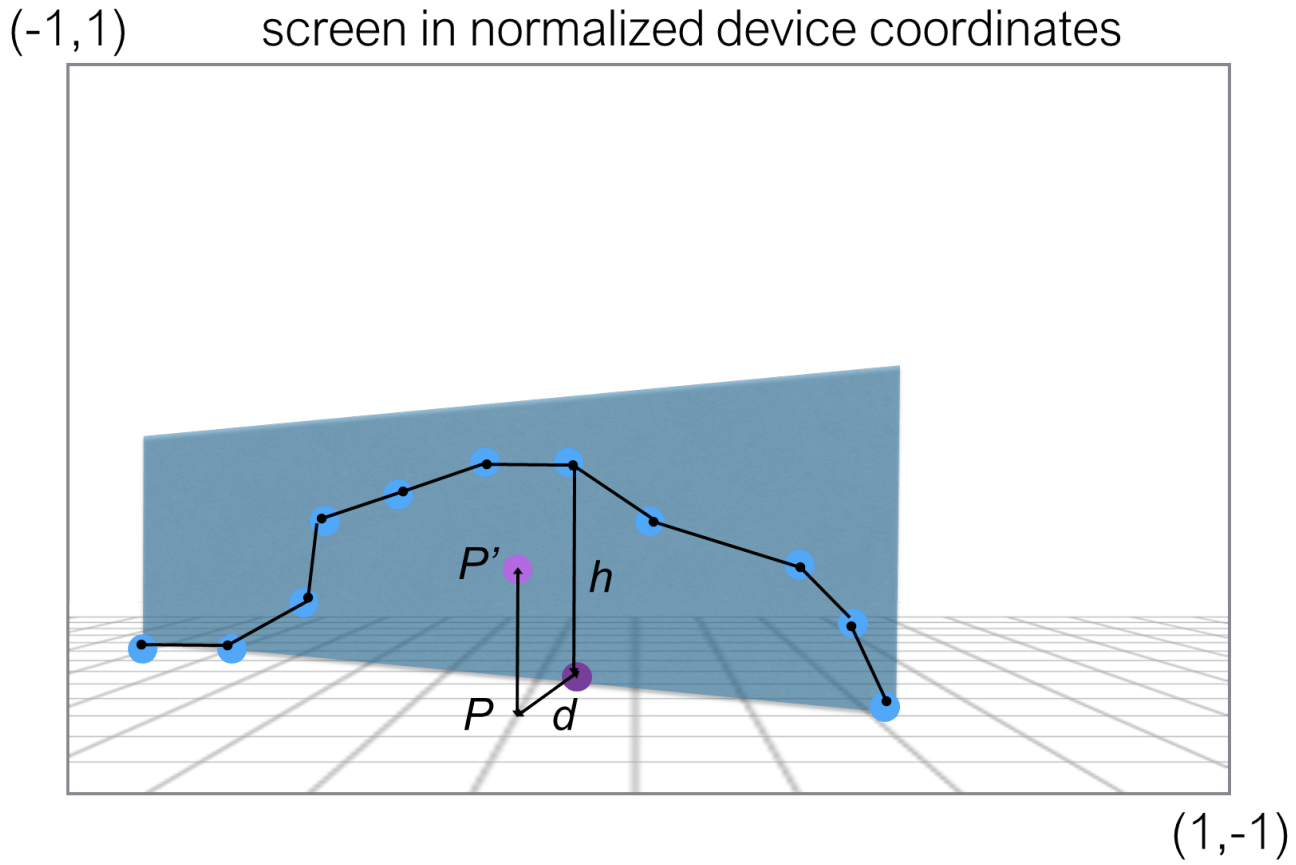
For each vertex **P** in the ground mesh:

- Find the **closest point** to **P** that lies within the projection plane.
- Find **h**, the height of the silhouette curve relative to **P_in_plane**.
- Find **d**, the distance from **P** to the projection plane

Terrain Editing Algorithm Step 3

$$P'_y = \begin{cases} (1 - w(d)) \cdot P_y + w(d) \cdot h & \text{if } h \neq 0 \\ P_y & \text{if } h = 0 \end{cases}$$

$$w(d) = \max\left(0, 1 - \left(\frac{d}{5}\right)^2\right).$$

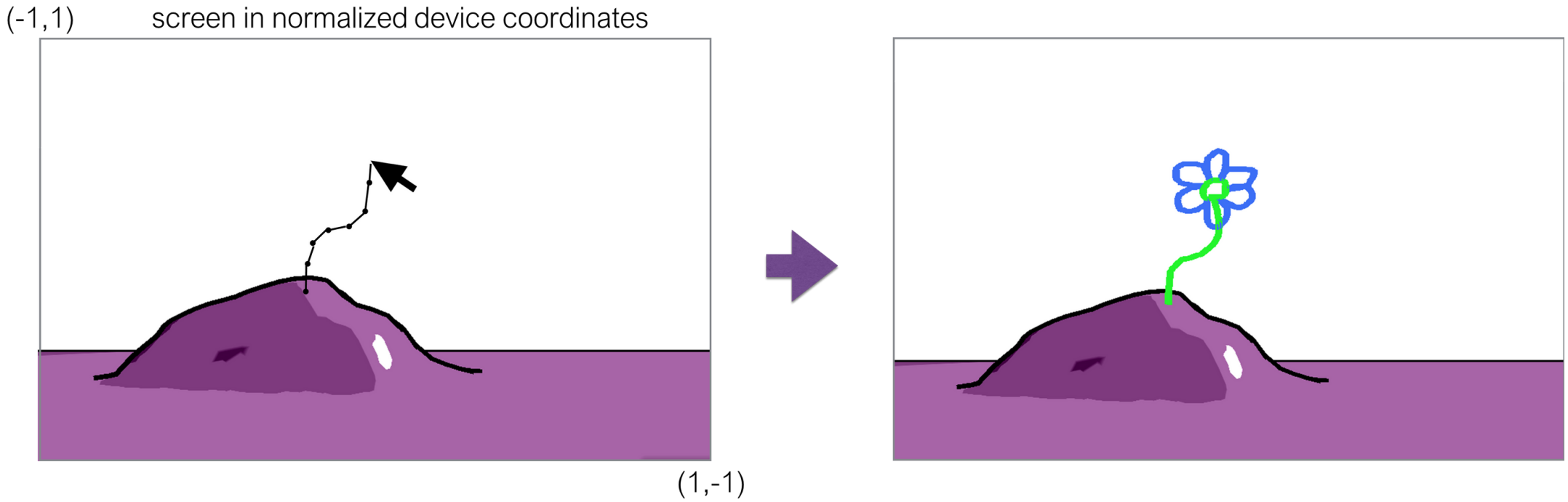


For each vertex **P** in the ground mesh:

- Find the **closest point** to **P** that lies within the projection plane.
- Find **h**, the height of the silhouette curve relative to **P_in_plane**.
- Find **d**, the distance from **P** to the projection plane

  Adjust **P**'s height based on **h**.

Feature #3: Drawing Billboards



This feature is already fully implemented in the assignment starter code.

Feature #3: Drawing Billboards

Like the sky strokes, we can start with the mesh in normalized device coordinates that is defined in 2D and then project this into 3D space.

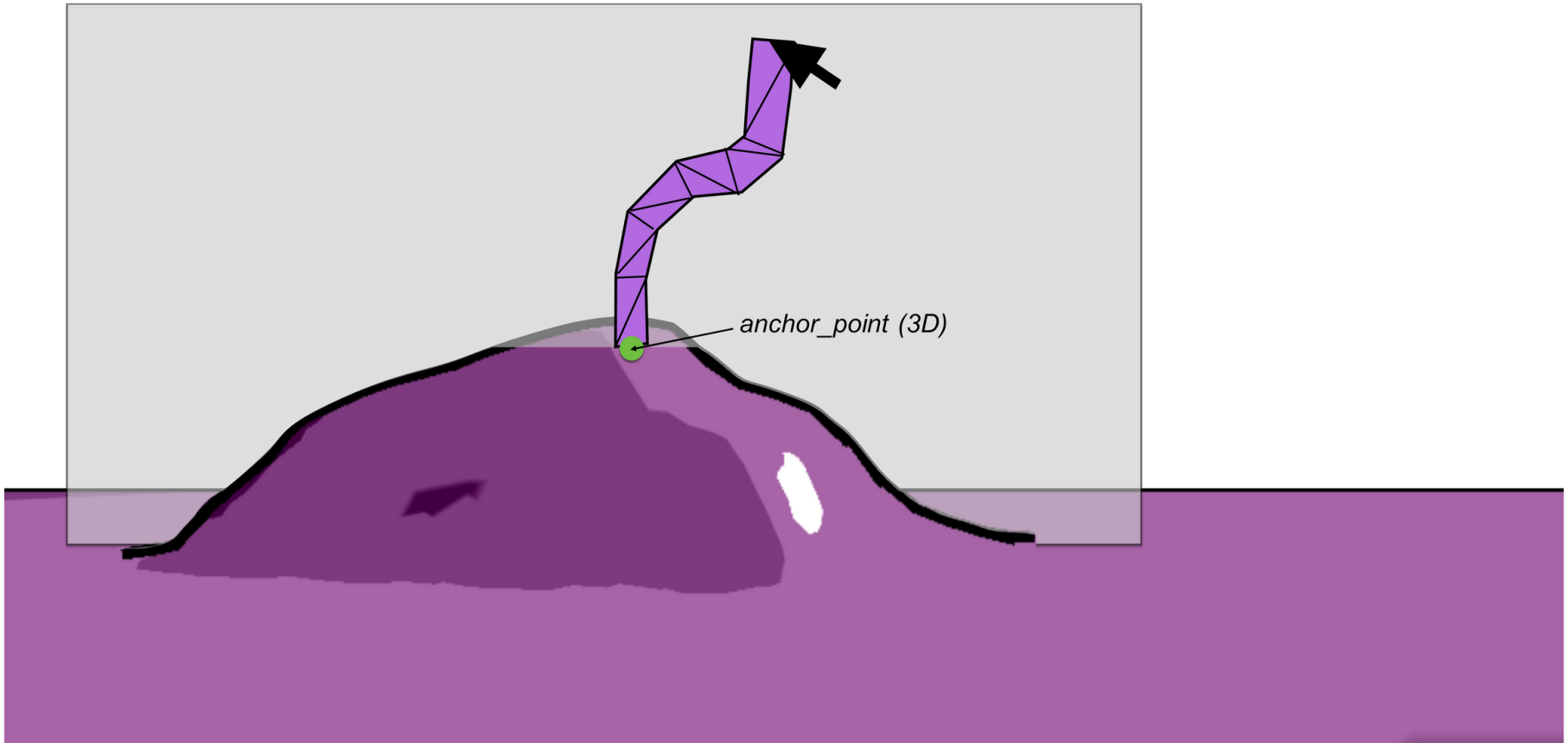
But what should we project onto this time?

Feature #3: Drawing Billboards

We can define a plane in 3d-space with a point that lies within the plane and a normal.

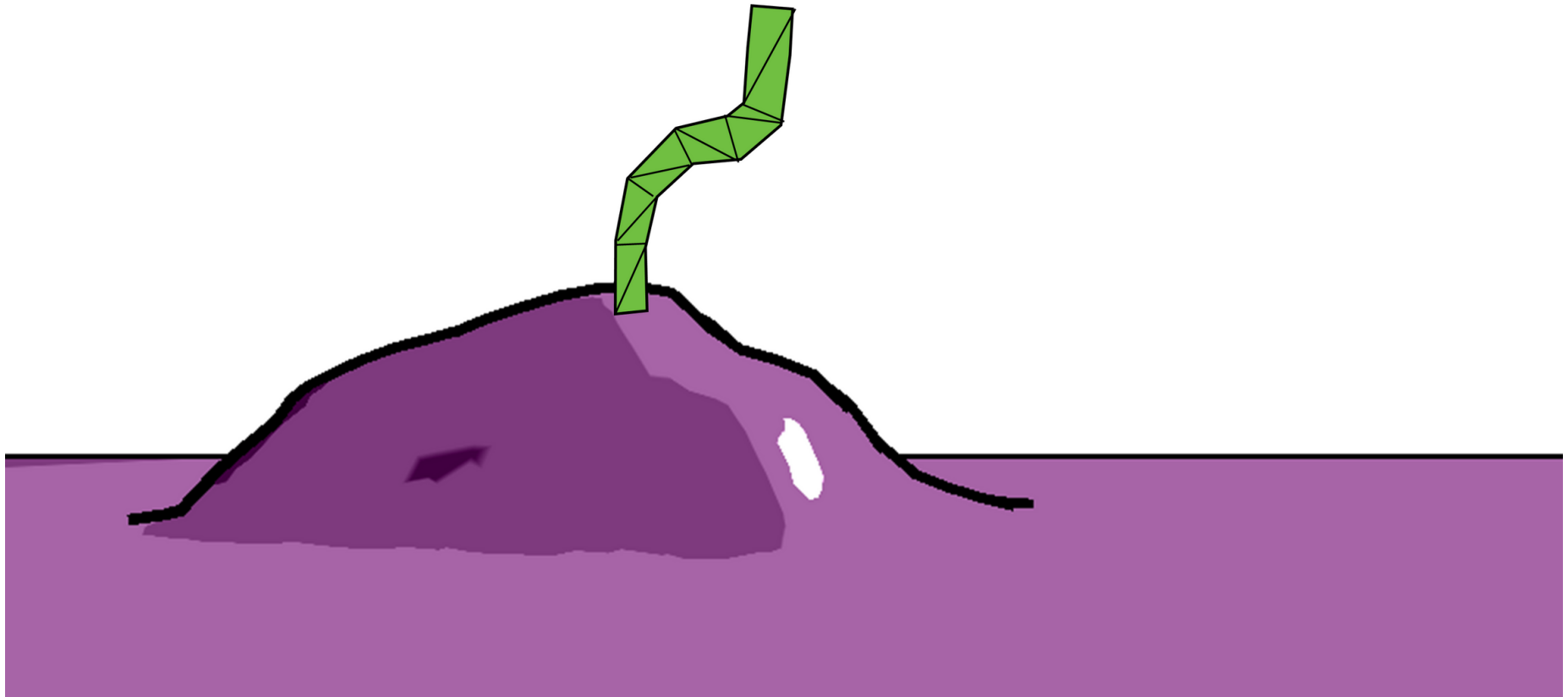
We'll use the first point of the stroke, which we know must intersect the ground as the **anchor_point**.

What should we use for the normal of this plane?

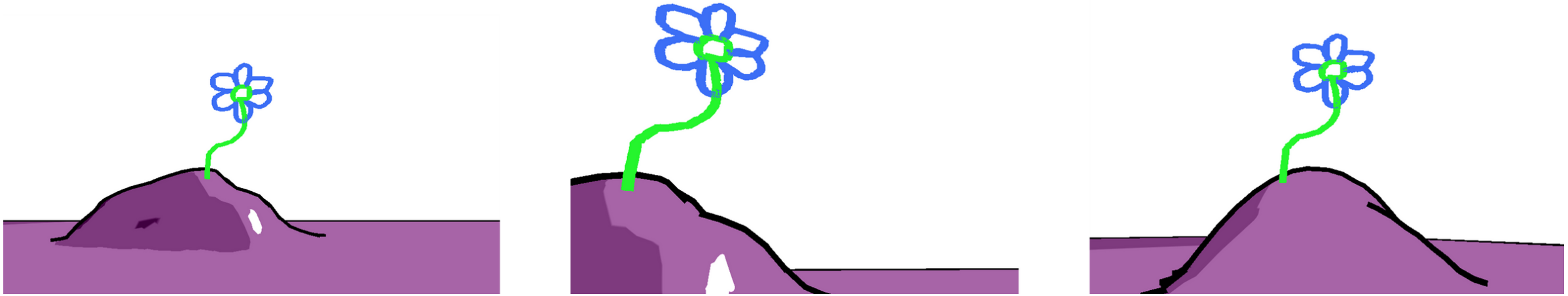


Feature #3: Drawing Billboards

Once we have defined a plane to project onto, creating a 3D triangle mesh based on the original 2D triangles follows a similar approach as for the sky.

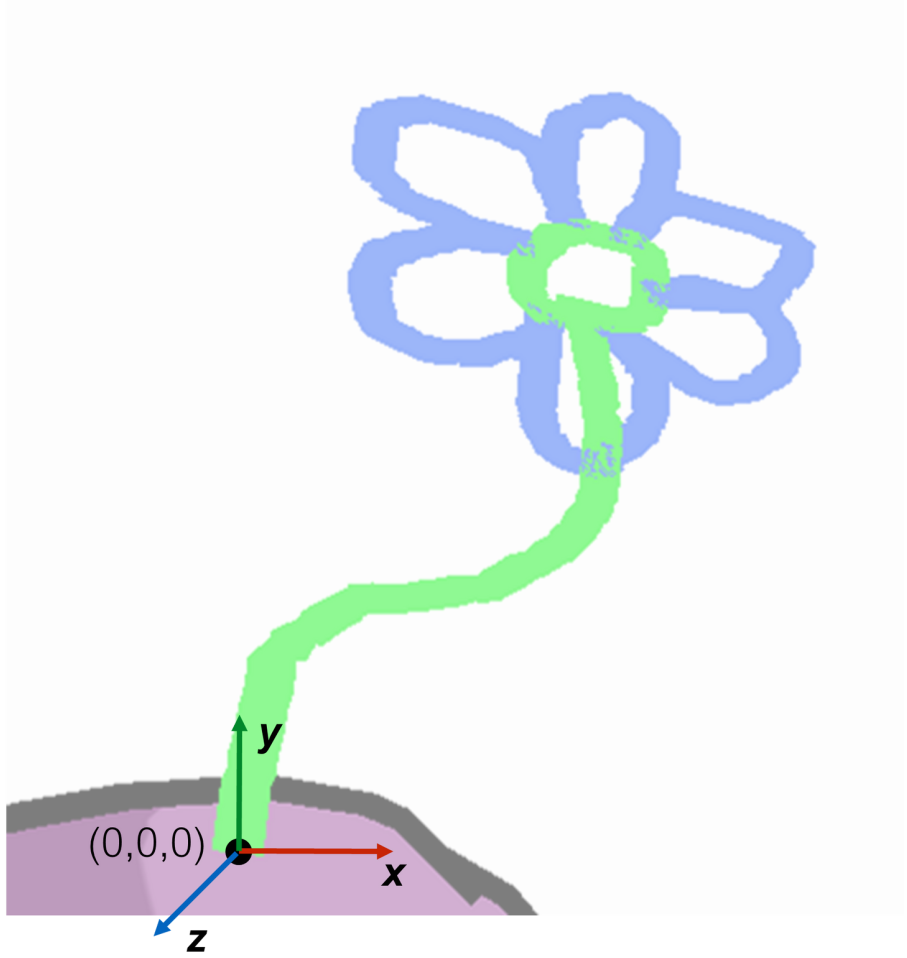


How to Actually Render the Billboards?



In computer graphics, a **billboard** is a 2D planar object that rotates to face the camera as the viewer moves around the 3D scene.

Billboard Coordinate System and Transformations



billboard local coordinate system

Before saving the 3D mesh for the billboard, transform its vertices so that they are defined within a more convenient *billboard coordinate system*.

- **origin** = billboard's anchor point
- **-z** = billboard plane normal

When updating the scene, transform the billboard object as follows:

- translate to the anchor point

  rotate the object to lookAt() the camera