9 min read · Apr 3, 2019

Michel de Brisis Follow

One of our clients at Apt got excited about a demo we made showcasing off-axis projection. So we made an installation using this technique, with headtracking from the Kinect v2. We posted a short video clip of the installation on reddit, and there seemed to be a fairly large demand for an article about how to make this work.

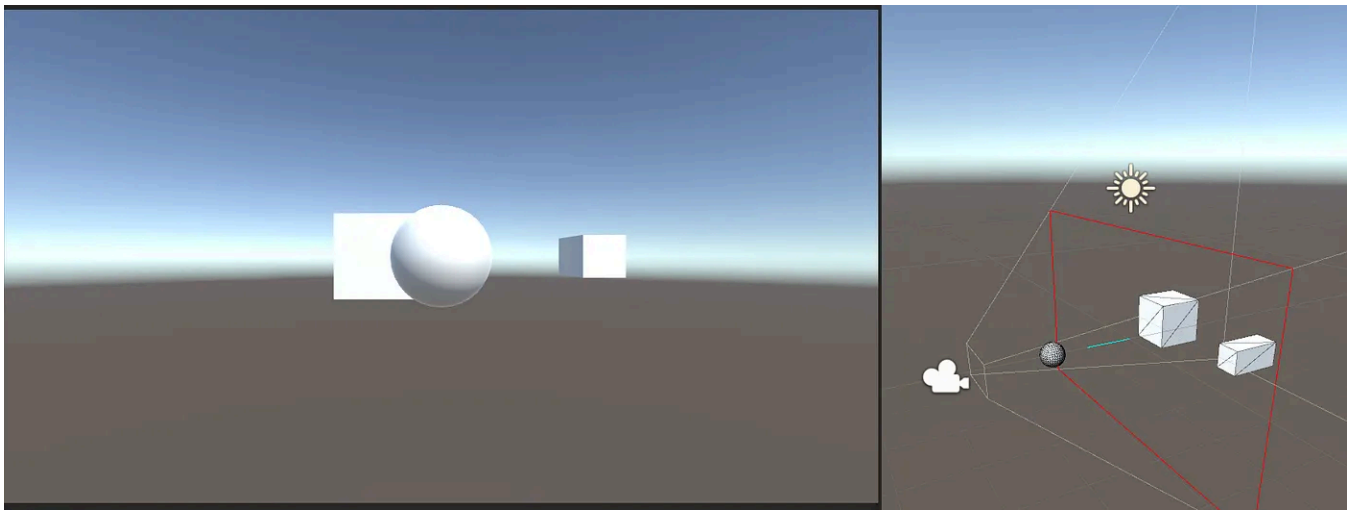
Progress shot in unity

Our installation

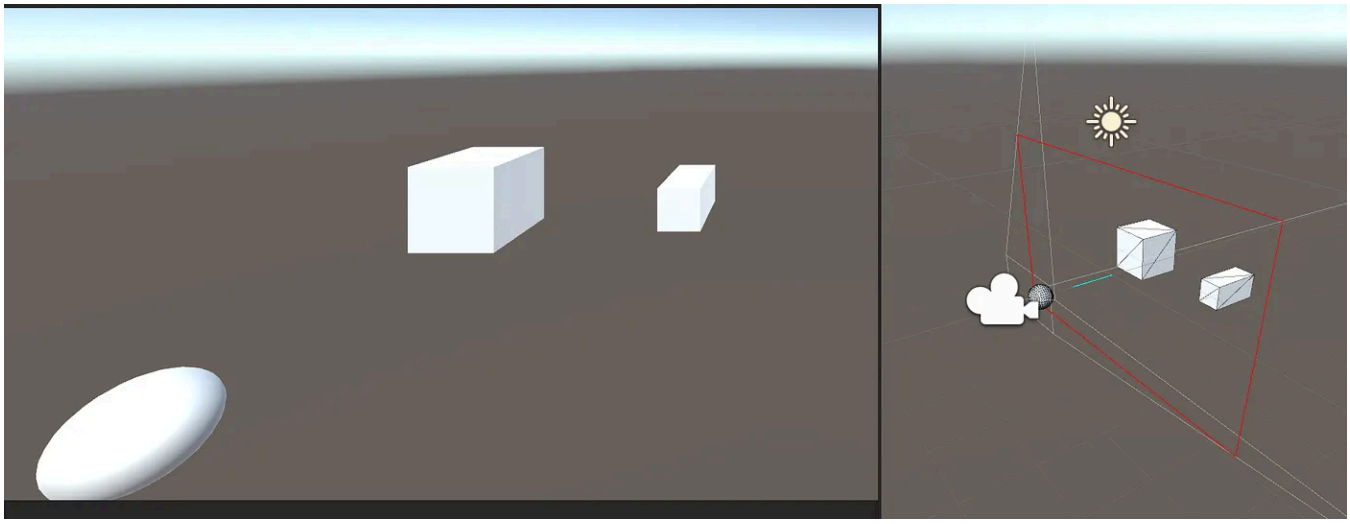
We used Unity, but the technique used is general purpose and can be adapted to pretty much any 3D rendering framework.

I based my code heavily on an article by Robert Kooima called Generalized Perspective Projection. I suggest you read that article first, to get a general sense of how off-axis projection works and for some in-depth explanation of some of the code to follow.

Alright, let's get to it! I recommend downloading my unity project from github so you can follow along in Unity (makes it a lot easier to visualize what's happening). For those of you who can't be bothered / don't use Unity, here are a few screenshots showing off the effect.



Camera view head on (on-axis)

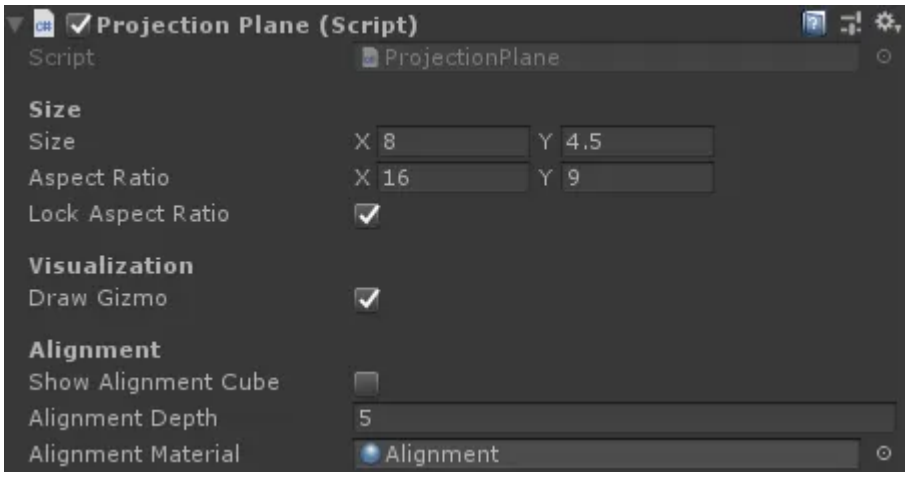


Camera moved up right, so skewing is visible (off-axis)

Notice how, in the bottom image, the geometry of objects skews. This is the effect we want. Normally, we view a monitor head on, so on-axis projection makes the most sense. However, try moving your head above and to the right of the monitor, and look at the bottom image again (You might have to get quite close to the monitor for the effect to work). Suddenly the geometry looks normal again, because the projection compensates for your viewing position. So how can we achieve this?

The Projection Plane

First up I wanted a convenient way to visualise and modify the projection plane we're viewing through (a virtual screen/window). I created the `ProjectionPlane` component to make it easier to separate the camera code from the projection plane. This class is annotated with the [ExecuteInEditMode] attribute, so we can see what going on in the editor. I exposed some properties:



Projection Plane properties

Let's focus on the Size and Aspect Ratio for now. The off-axis projection only looks correct when it matches the aspect ratio of the monitor it is being displayed on. The following code runs in the `Update()` function, so that we can set the size of the plane we're viewing through, with some aspect ratio locking thrown in for convenience.

```
if(LockAspectRatio)
{
    if(AspectRatio.x != previousAspectRatio.x)
    {
        Size.y = Size.x / AspectRatio.x * AspectRatio.y;
        //if both change, X takes precedence
        previousAspectRatio.y = AspectRatio.y;
    }

    if(AspectRatio.y != previousAspectRatio.y)
    {
         Size.x = Size.y / AspectRatio.y * AspectRatio.x;
    }

    if(Size.x != previousSize.x)
    {
         Size.y = Size.x / AspectRatio.x * AspectRatio.y;
         // if both change, X takes precedence
         previousSize.y = Size.y;
    }

    if(Size.y != previousSize.y)
    {
        Size.x = Size.y / AspectRatio.y * AspectRatio.x;
    }
}

 //Make sure we don't crash unity
 Size.x = Mathf.Max(1, Size.x);
 Size.y = Mathf.Max(1, Size.y);
```

```
    AspectRatio.x = Mathf.Max(1, AspectRatio.x);
    AspectRatio.y = Mathf.Max(1, AspectRatio.y);

    previousSize = Size;
    previousAspectRatio = AspectRatio;
```

You can definitively crash Unity by setting the values of either `Size` or `AspectRatio` too low (effectively breaking the projection matrix later on), so I put in a little guard for that.

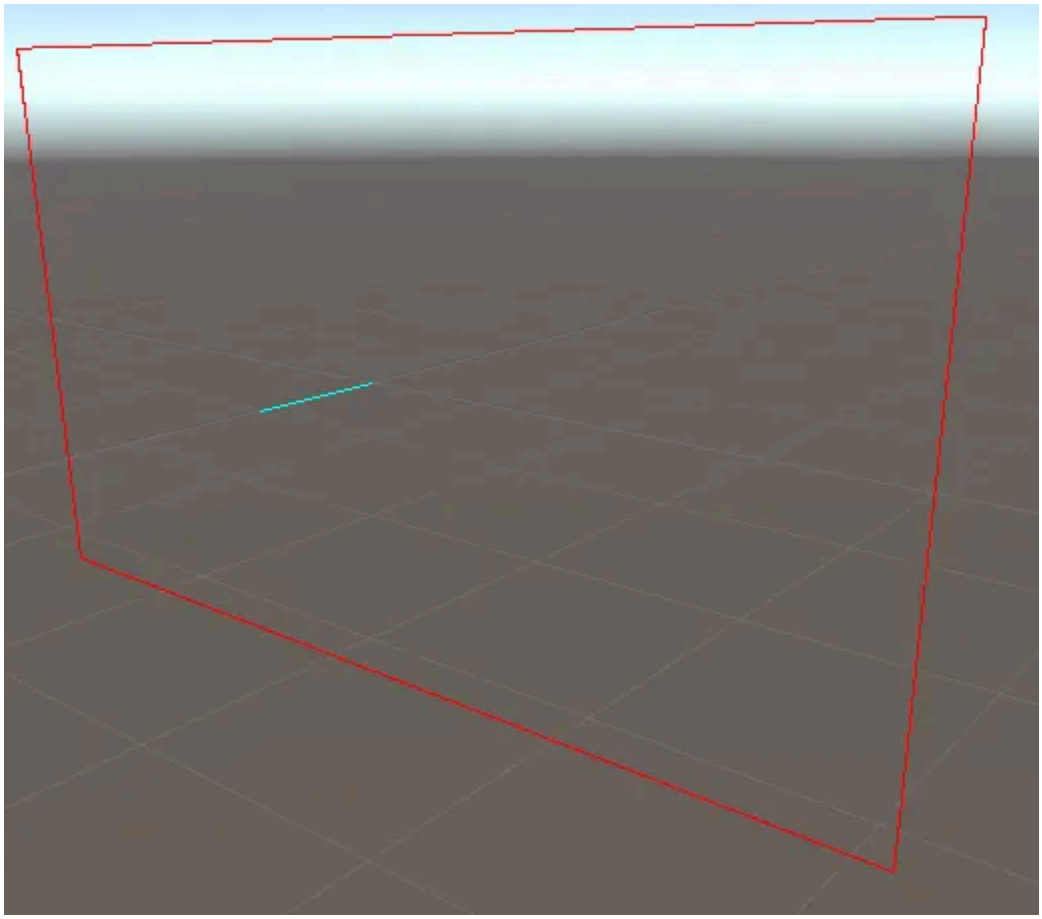
Next up are the vectors and matrix we need to pass on to the camera (also in the `Update()` function)

```
    BottomLeft = transform.TransformPoint(
        new Vector3(-Size.x, -Size.y) * 0.5f);
    BottomRight = transform.TransformPoint(
        new Vector3(Size.x, -Size.y) * 0.5f);
    TopLeft = transform.TransformPoint(
        new Vector3(-Size.x, Size.y) * 0.5f);
    TopRight = transform.TransformPoint(
        new Vector3(Size.x, Size.y) * 0.5f);

    DirRight = (BottomRight - BottomLeft).normalized;
    DirUp = (TopLeft - BottomLeft).normalized;
    DirNormal = -Vector3.Cross(DirRight, DirUp).normalized;

    m = Matrix4x4.zero;
    m[0, 0] = DirRight.x;
    m[0, 1] = DirRight.y;
    m[0, 2] = DirRight.z;

    m[1, 0] = DirUp.x;
    m[1, 1] = DirUp.y;
    m[1, 2] = DirUp.z;

    m[2, 0] = DirNormal.x;
    m[2, 1] = DirNormal.y;
```

```
m[2, 2] = DirNormal.z;

m[3, 3] = 1.0f;
```

I'm basically following Kooima's article for setting up the vectors and matrix we'll need in the camera code later. Note that I transform the points from local space to world space. Also note that I calculate all four bounds, which I use in the Gizmo visualisation below (in `OnDrawGizmos()`)

```
Gizmos.color = Color.red;
Gizmos.DrawLine(BottomLeft, BottomRight);
Gizmos.DrawLine(BottomLeft, TopLeft);
Gizmos.DrawLine(TopRight, BottomRight);
Gizmos.DrawLine(TopLeft, TopRight);

//Draw direction towards eye
Gizmos.color = Color.cyan;
var planeCenter = BottomLeft + ((TopRight - BottomLeft) * 0.5f);
Gizmos.DrawLine(planeCenter, planeCenter + DirNormal);
```



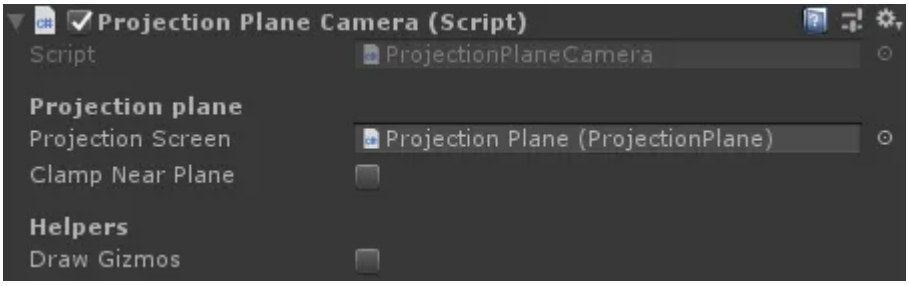
I've also added a little line that is drawn towards the side of the plane where the camera should be.

The Projection Camera

Next up is the camera class.

```
[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
public class ProjectionPlaneCamera : MonoBehaviour
```

We need to be able to see how the camera behaves in the editor, hence the `[ExecuteInEditMode]` . We also **really** need a Camera to show something on screen, hence the `[RequireComponent(typeof(Camera)]` attribute. We then expose some properties



ProjectionPlaneCamera properties

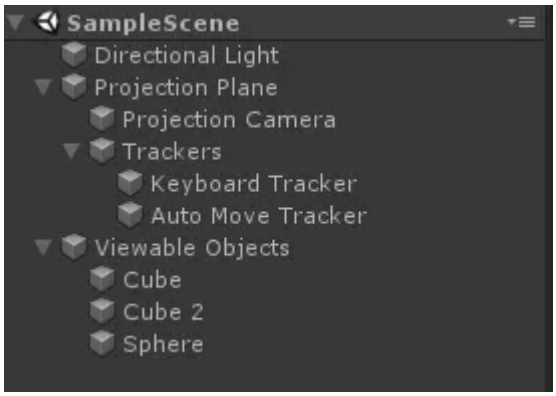
We need a reference to the `ProjectionPlane` which is the basis for our camera's custom projection and worldToCamera matrices. And then we get to the meat of it, the custom matrices which are set up in the `LateUpdate()` function.

```
if(ProjectionScreen != null)
{
    Vector3 pa = ProjectionScreen.BottomLeft;
    Vector3 pb = ProjectionScreen.BottomRight;
    Vector3 pc = ProjectionScreen.TopLeft;
    Vector3 pd = ProjectionScreen.TopRight;

    Vector3 vr = ProjectionScreen.DirRight;
    Vector3 vu = ProjectionScreen.DirUp;
    Vector3 vn = ProjectionScreen.DirNormal;

Matrix4x4 M = ProjectionScreen.M;

    eyePos = transform.position;
```

```
        //From eye to projection screen corners
        va = pa - eyePos;
        vb = pb - eyePos;
        vc = pc - eyePos;
        vd = pd - eyePos;

        viewDir = eyePos + va + vb + vc + vd;

        //distance from eye to projection screen plane
        float d = -Vector3.Dot(va, vn);
        if (ClampNearPlane)
        cam.nearClipPlane = d;
        n = cam.nearClipPlane;
        f = cam.farClipPlane;


        float nearOverDist = n / d;
        l = Vector3.Dot(vr, va) * nearOverDist;
        r = Vector3.Dot(vr, vb) * nearOverDist;
        b = Vector3.Dot(vu, va) * nearOverDist;
        t = Vector3.Dot(vu, vc) * nearOverDist;
        Matrix4x4 P = Matrix4x4.Frustum(l, r, b, t, n, f);

        //Translation to eye position
        Matrix4x4 T = Matrix4x4.Translate(-eyePos);


        Matrix4x4 R = Matrix4x4.Rotate(
            Quaternion.Inverse(transform.rotation) *
    ProjectionScreen.transform.rotation);

        cam.worldToCameraMatrix = M * R * T;

        cam.projectionMatrix = P;
    }
```

Again, I'm just implementing code from Kooima's article, and throwing in a viewing direction vector for visualising what the focal point in the scene will be. By setting the `cam.worldToMatrix` we ensure that shadow calculations will be correct, and as an added bonus, the default frustum Gizmo for the camera works as intended. The ClampNearPlane flag sets whether or not we clamp the camera's near plane to be exactly the same as the projection plane. Most times that's the most useful near plane. However, if you want to do some stereoscopic effect where things pop out of the screen, you can turn off the clamping and place geometry between the camera and the projection plane. This geometry will behave as if between the user and the screen, but only if you have a working stereoscopic method for displaying it.

Setting up the hierarchy



Scene hierarchy

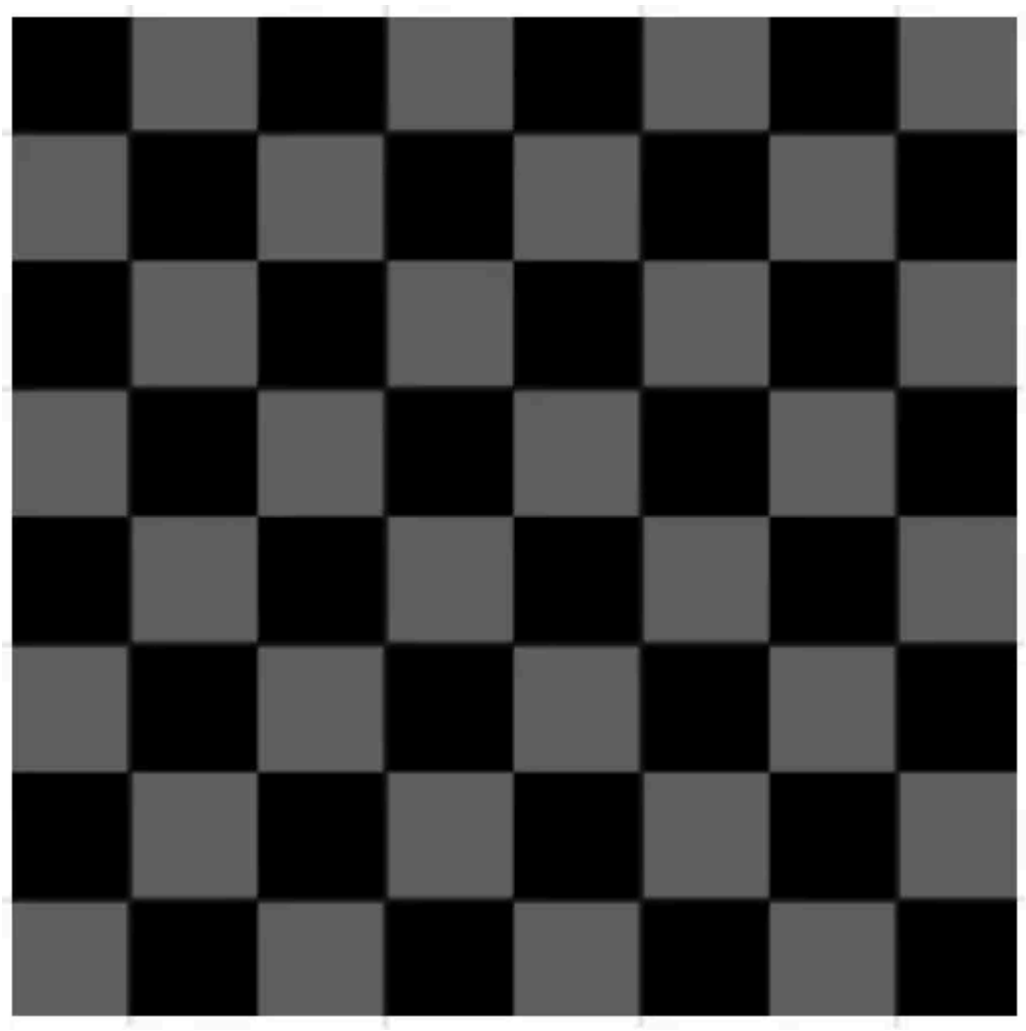
For convenience I set up the camera to be a child of the projection plane. This has the benefit of being able to move the camera relative to the plane, and you can move the projection plane around with the camera attached. It's a good rule of thumb to move the camera to a default position by only offsetting it in the Z direction from the projection plane (along the helper line on the Projection Plane gizmo). That way, the default view (before applying movement to the camera) is an on-axis projection.

Calibration and the alignment cube

You might have noticed in the projection plane properties that I created a small section called *Alignment* with references to an alignment cube. This alignment cube is created and hidden when the application starts, unless the **Show Alignment Cube** flag is set. The code for setting it up resides in `Start()` in the `ProjectionPlane` class.

```
if(Application.isPlaying)
{
    alignmentCube = new GameObject("AlignmentCube");
    alignmentCube.transform.SetParent(transform, false);

    alignmentCube.transform.localPosition = Vector3.zero;
    alignmentCube.transform.rotation = transform.rotation;

    GameObject back = CreateAlignmentQuad();
    backTrans = back.transform;
    GameObject left = CreateAlignmentQuad();
    leftTrans = left.transform;
    GameObject right = CreateAlignmentQuad();
    rightTrans = right.transform;
    GameObject top = CreateAlignmentQuad();
    topTrans = top.transform;
    GameObject bottom = CreateAlignmentQuad();
    bottomTrans = bottom.transform;

}
```

This just creates 5 quads and stores their transforms. I also set up the material as an unlit texture with the following texture (It's worth noting that there's a white border around the grid)

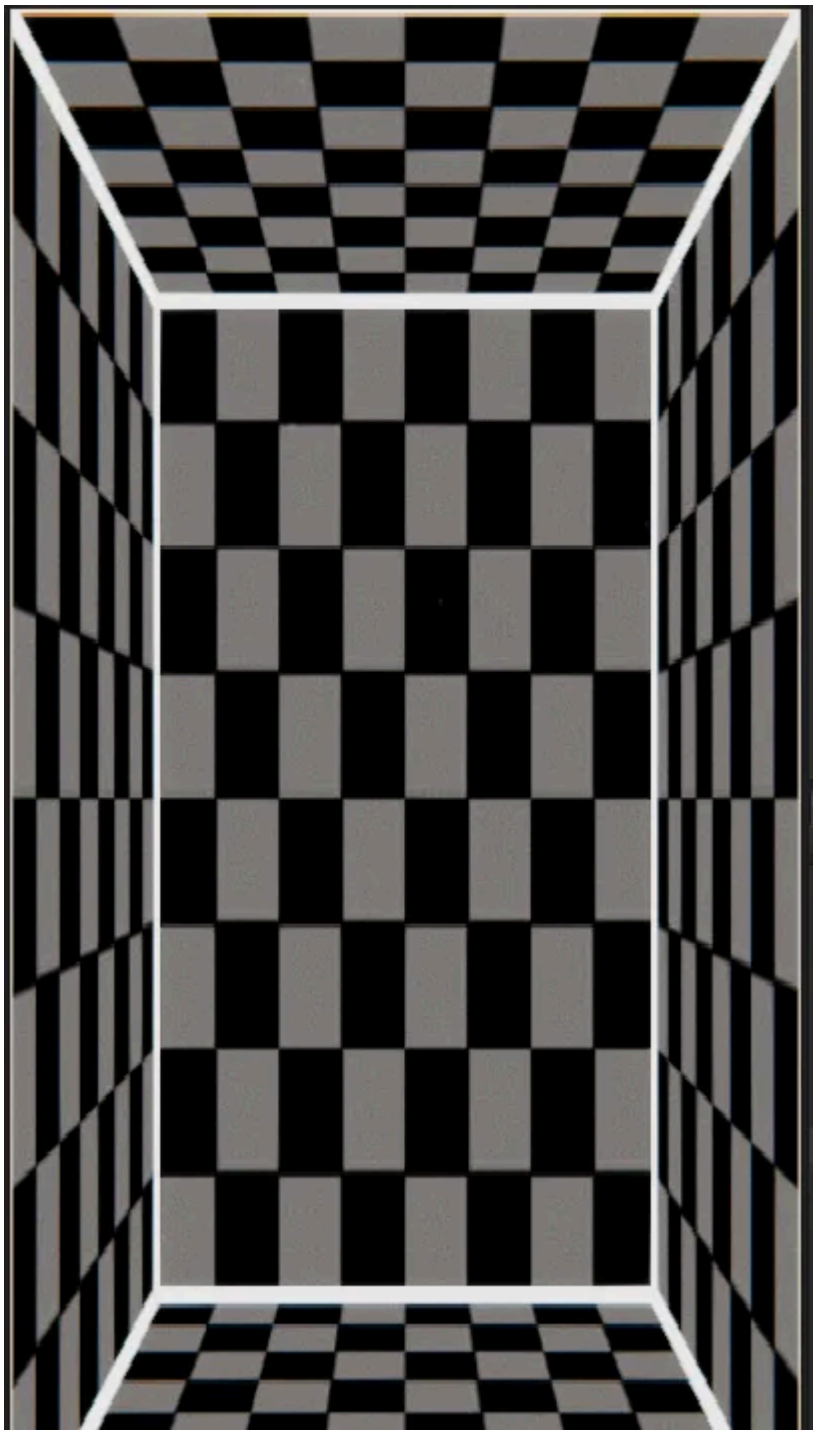


The scaling of the Alignment cube happens every update to make sure you can resize the projection plane and still have the cube aligned to the camera view. The following code runs in `Update()`

```
public void UpdateAlignmentCube()
{
    Vector2 halfSize = Size * 0.5f;
    UpdateAlignmentQuad(backTrans,
        new Vector3(0, 0, AlignmentDepth),
        new Vector3(Size.x, Size.y), Quaternion.identity);
    UpdateAlignmentQuad(leftTrans,
       new Vector3(-halfSize.x, 0, AlignmentDepth * 0.5f),
       new Vector3(AlignmentDepth, Size.y, 0),
       Quaternion.Euler(0, -90, 0));
    UpdateAlignmentQuad(rightTrans,
        new Vector3(halfSize.x, 0, AlignmentDepth * 0.5f),
        new Vector3(AlignmentDepth, Size.y, 0),
```

```
            Quaternion.Euler(0, 90, 0));
        UpdateAlignmentQuad(topTrans,
            new Vector3(0, halfSize.y, AlignmentDepth * 0.5f),
            new Vector3(Size.x, AlignmentDepth, 0),
            Quaternion.Euler(-90, 0, 0));

        UpdateAlignmentQuad(bottomTrans,
            new Vector3(0, -halfSize.y, AlignmentDepth * 0.5f),
            new Vector3(Size.x, AlignmentDepth, 0),
            Quaternion.Euler(90, 0, 0));
    }

    private void UpdateAlignmentQuad(Transform t, Vector3 pos, Vector3
    scale, Quaternion rotation)
    {
        t.localPosition = pos;
        t.localScale = scale;
        t.localRotation = rotation;
    }
```

So what do we use this cube for? Well, it's helpful for trackers (which we'll touch on next) which might have an offset. For our installation, we attached 4 rods to the corners of the monitor, so that when you see the alignment cube from various angles, the perspective lines should always line up with corner rods pointing directly out of the monitor (like normals to the monitor).



The alignment cube in action

Trackers and camera movement

To test the camera movement and projection I added a base class for trackers which looks like this

```
public class TrackerBase : MonoBehaviour
{
    [HideInInspector]
    public bool IsTracking { get; protected set; }
    [HideInInspector]
    public ulong TrackedId { get; protected set; }
    [HideInInspector]
    public Vector3 Translation { get => translation; }
```

```
    [HideInInspector]
    public float SecondsHasBeenTracked { get; protected set; }

    protected Vector3 translation;
}
```

I also implemented a quick'n'dirty movement of the camera for testing with the following class

```
public class AutoMoveTracker : TrackerBase
{
    [Min(0.001f)]
    public float BoundsSize = 2;

    [Range(0, 1)]
    public float XMovement = 0.5f;
    [Range(0, 1)]
    public float YMovement = 0.3f;
    [Range(0, 1)]
    public float ZMovement = 0;

    private float HalfBoundSize => BoundsSize * 0.5f;

    void Start()
    {
        IsTracking = true;
    }

    void Update()
    {
       if(Input.GetKeyUp(KeyCode.A) &&
(Input.GetKey(KeyCode.LeftControl) ||
Input.GetKey(KeyCode.RightControl)))
       {
           IsTracking = !IsTracking;
           SecondsHasBeenTracked = 0;
       }

       if(IsTracking)
       {
           SecondsHasBeenTracked += Time.deltaTime;
           float xSize = XMovement * HalfBoundSize;
           float ySize = YMovement * HalfBoundSize;
           float zSize = ZMovement * HalfBoundSize;
           translation.x =  Mathf.Sin(SecondsHasBeenTracked) *
xSize;
           translation.y =  Mathf.Sin(SecondsHasBeenTracked -
(Mathf.PI * 2 / 3)) * ySize;
           translation.z =  Mathf.Sin(SecondsHasBeenTracked) *
zSize;
       }
```

```
    }
}
```

Finally I created a small component which moves the camera based on the input from a BaseTracker and added it to my camera gameobject

```
[RequireComponent(typeof(ProjectionPlaneCamera))]
public class BasicMovement : MonoBehaviour
{
    public TrackerBase Tracker;

    private ProjectionPlaneCamera projectionCamera;
    private Vector3 initialLocalPosition;

    void Start()
    {
        projectionCamera = GetComponent<ProjectionPlaneCamera>();
        initialLocalPosition =
projectionCamera.transform.localPosition;
    }

    void Update()
    {
        if (Tracker == null)
            return;

        if(Tracker.IsTracking)
         {
             projectionCamera.transform.localPosition =
initialLocalPosition + Tracker.Translation;
         }
    }
}
```

This movement manager has a lot of room for improvement, and the one used in the final version of the installation has a code for handling

- What happens when there is more than one person being tracked
- What happens when someone enters or leaves the tracking area
- Smoothing input

Caveats and gotchas

I'll mention a few gotchas I discovered along the way. Hopefully you'll disregard all my advice here and discover for yourself what happens when you do.

- Skyboxes — don't use them. Use geometry in the scene for backgrounds.
- Save yourself the headache, do not scale the `ProjectionPlane` transform. It messes with the local coordinate space of the camera, making the tracking unneccessarily complex to handle.
- Custom projection matrices currently do not play well with the new Postprocessing v2 stack. Check out https://docs.unity3d.com/ScriptReference/Camera-nonJitteredProjectionMatrix.html for more information

Game Development | Unity3d | Projection | Code | Csharp

Written by Michel de Brisis

Developer of games, installations, and all other things fun