Scott Lembcke, Blogger
August 25, 2017

12 Min Read

For our game, Verdant Skies, we wanted it to have a traditional hand drawn look, but we also wanted to make it more interesting by giving it a full 3D perspective and adding effects such as shadows and water reflections. This fit the skills of our small team well.

On older, 2D hardware, it would certainly be possible to achieve a similar look simply by controlling the drawing order, calculating the position and scale of each sprite. Since we were using a 3D engine (Unity) we might as well save ourselves some trouble! With the default settings, Unity won't help you make a projection like this, and there are a few issues. First of all, Unity doesn't really know how to depth sort sprites in a scene like this. Secondly, the projection needed is not a basic perspective like the one provided by Unity, and it can look quite bad if you use one. However, with the right settings and a little matrix math, it is possible to get a classic projection without a lot of work. For an example, try our interactive WebGL demo. A link to our camera script is included at the end of the article in the resources section.

Drawing Order

The first issue is how to draw the sprites in the correct order. Since most 2D games use alpha blending, Unity must draw the sprites in the right order without the benefit of z-buffers. Unity's sorting support is somewhat limited for 2D games, and people have tried a lot of creative, though tedious attempts to work around it. Fortunately, though with a combination of several features a simple solution exists.

Since Verdant Skies uses a 3D perspective, it wasn't possible to use z-value exclusively for depth sorting. Moving objects along the z-axis would change their size and break their reflections and shadows. Also, because Unity depth sorts sprites by their center, we couldn't rely on the camera to sort Verdant Skies' sprites automatically either as some sprites laid on the ground and others stand upright. If the player walked past the middle of a patch on the ground, their sprite would disappear underneath the ground! Using sorting layers and orders isn't ideal, because it requires updating the sorting order every time an object moves. Also, the sorting order property provides very little precision (only 16 bits), which limits the size of the world.

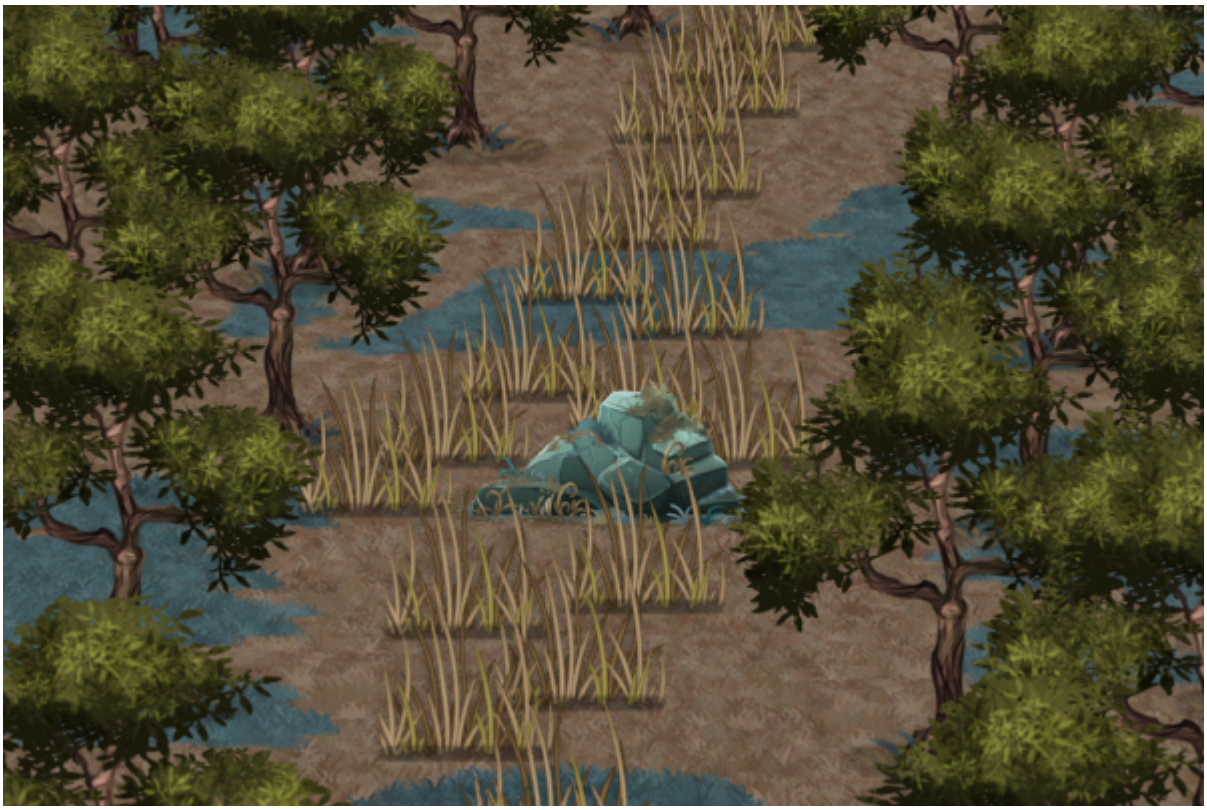
Instead, we use sorting layers to draw sprites from the ground up, and use the orthographic camera sorting mode to depth sort sprites within the same layer. The basic ground patches were drawn first, then the detail patches on top (gravel, fields, etc), and then finally the sprites that stood upright. Since ground patches are always flat, plants and characters are always upright, etc, then the sorting layer never changes and becomes a simple matter of setup.

The remaining task is to sort sprites within the same layer. This really only matters for upright objects, and we want them to be sorted automatically by their depth in the scene. Orthographic cameras already default to the correct sorting mode. However, for perspective cameras the default sorting mode compares objects by their distance to the camera's center. This doesn't work with sprites since moving the camera left or right will cause them to pop in front on one another. It's especially noticeable for large sprites like buildings or trees and can be very distracting. Fortunately, you can configure your perspective camera to sort by depth like orthographic cameras do. You can simply configure your camera in a *Start()* method.

```
camera.transparencySortMode = TransparencySortMode.Orthographic;
```

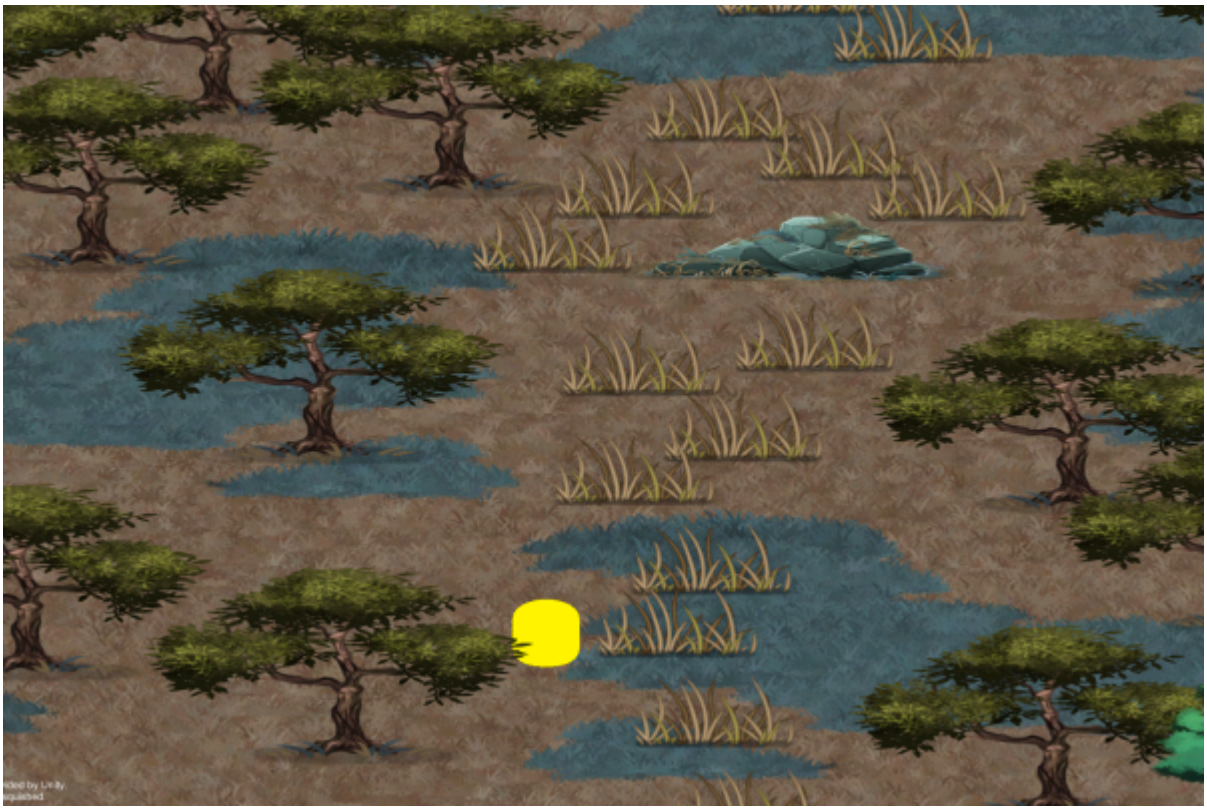
Put together, the sorting layer draws sprites from the ground up and the camera's depth sorting draws them from front to back. All of the sorting will happen automatically without needing to intervene with your own scripts.

2D Projections



The next task is to render the correct projection. Consider screenshot above as the goal. The sprites are all drawn parallel to the screen without any distortion. The ground is viewed from a 30° angle with a 60° FOV to give it some subtle perspective. (This is hard to see in a static image, but try the demo above.) Ideally it should be possible to able to achieve this look without writing a lot of code, or changing the all the objects in the scene to make the camera work.

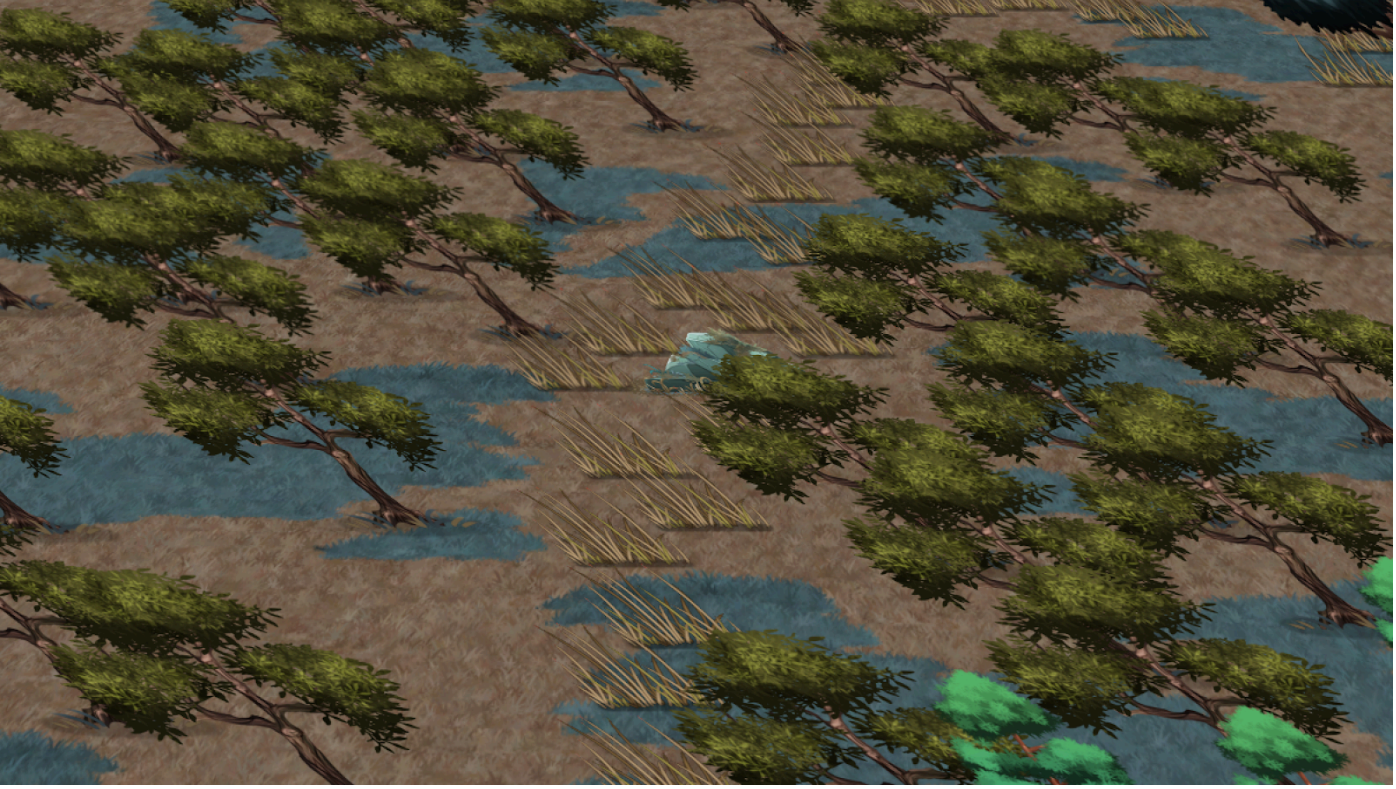
The most obvious solution is probably to try a perspective camera, but the screenshot above shows the problem. Even with a modest 60° field of view, the camera is looking directly down on the sprites near the bottom of the screen, revealing their paper thin existence. On the other hand, the trees near the top do look ok.



An orthographic projection doesn't fare much better. Although it does a better job hiding that the art is just a flat facade, it now distorts all of the sprites the same amount and they all look squished! A potential fix for both of these basic projections is that if we tilted all of the sprites so they pointed at the camera, then it would probably look good. In fact, that is exactly what the fix is. Since tilting every sprite in every scene would be tedious, math to the rescue!

In all of these cases, we are looking at mappings from one coordinate space (3D coordinates) to another (screen coordinates). That is the essence of a projection, and there are quite a few useful ones other than the two that Unity provides. You've likely heard of two or three point perspective, orthographic, and isometric projections. These are examples of linear perspectives, but there are also plenty of non-linear projections too, such as fisheye lenses and all of those crazy map projections you've seen of the Earth.

Since realtime 3D graphics is very heavily based around matrices, it's perhaps no surprise then that the two basic projections offered by Unity can be represented by a matrix. In fact, any linear projection can be reduced to a matrix, so the interesting question to ask is which projections are linear. The easiest way to think about it is: if something is a straight line in the world, then it will be a straight line in the projection. That rule works both ways too, so any straight line on the screen is projected from a straight line in the world as well. Given the right matrix, we should be able to get Unity to render any linear projection we want, even Ultima's unique (and sometimes despised) oblique projection!



While this article is about fixing Unity's projection for 2D content, it also works with 3D content, and is used for similar reasons. In fact, A Link Between Worlds used this same technique to more closely match the look of older 2D Zelda games. The game was even rendered in stereoscopic 3D without causing any obvious distortion for the viewer.

Top-down view

A view from the side reveals the trick

(via Zelda Dungeon)

Using Custom Projections with Unity

In basically all 3D engines, a matrix is used to transform objects from the world onto the screen. This matrix is called the model-view-projection matrix, and is broken into three parts. First, the model matrix transforms from model coordinates to world coordinates (*Transform.localToWorldMatrix*). Each object gets its own model matrix that positions it relative to other objects based on the its transform. Next, the view matrix transforms world coordinates to be relative to the camera (*Camera.worldToCameraMatrix*). Finally the projection matrix transforms those camera relative coordinates into screen coordinates (*Camera.projectionMatrix*). The

view and projection matrix are shared by all objects in a scene and determine the overall projection to the screen. Usually, the projection matrix is only responsible for choosing between an orthographic or perspective projection and the overall scale of the scene as rendered by the camera. Think of it like a camera's lens.

To set up a custom projection, first choose if you want an orthographic or perspective view. The projection matrix handles that part, and you can use Unity's regular Camera inspector to set it up. The direction each axis points relative to the screen are then handled by the view matrix. Normally, Unity updates this matrix every frame based on the camera's transform, but writing to the *Camera.worldToCameraMatrix* property overrides it with a custom value. Since we are okay with how the ground looks, we don't need to change the output for the x or y-axes, and only need to change the z-axis. Specifically, we want the world's up direction to always agree with the camera's up direction. Since each column of the matrix corresponds to an axis, we just have to change the column for the z-axis. The code for that would look something like the following.

```
private void OnPreCull(){
    // First calculate the regular worldToCameraMatrix.
    // Start with transform.worldToLocalMatrix.
    var m = camera.transform.worldToLocalMatrix;
    // Then, since Unity uses OpenGL's view matrix conventions
    // we have to flip the z-value.
    m.SetRow(2, -m.GetRow(2));

    // Now for the custom projection.
    // Set the world's up vector to always align with the camera's up vector.
    // Add a small amount of the original up vector to
    // ensure the matrix will be invertible.
    // Try changing the vector to see what other projections you can get.
    m.SetColumn(2, 1e-3f*m.GetColumn(2) - new Vector4(0, 1, 0, 0));

    camera.worldToCameraMatrix = m;
}
```

The best place to set the view matrix is in the camera's *OnPreCull()* event method. This is called after all of the variations of the update methods, but before any of the rendering work begins.

User Input Coordinates

A game that uses mouse or touch input will likely need to convert screen coordinates into world coordinates. In a basic 2D game, *Camera.ScreenToWorldPoint()* would suffice, but when using a custom projection, it becomes more complicated. While it's possible to use existing Unity APIs to construct a ray and check it's intersection with the scene, if all you have is a ground plane, then there is a simpler way. Writing your own version of *ScreenToWorldPoint()* is only a few lines of code, and with only one more, you can make it work with coordinates on the ground plane. The basic idea is that Unity uses the view-projection matrix to convert world coordinates to the screen, so using the inverse of that matrix we can convert screen points to the world. By changing the matrix to ignore the z-axis, it's possible to ignore the scene depth and only get points on the ground plane.

```
public static Matrix4x4 ScreenToWorldMatrix(Camera cam){
    // Make a matrix that converts from
    // screen coordinates to clip coordinates.
    var rect = cam.pixelRect;
    var viewportMatrix = Matrix4x4.Ortho(rect.xMin, rect.xMax, rect.yMin, rec

    // The camera's view-projection matrix converts from world coordinates to
    var vpMatrix = cam.projectionMatrix*cam.worldToCameraMatrix;

    // Setting column 2 (z-axis) to identity makes the matrix ignore the z-ax
    // Instead you get the value on the xy plane!
    vpMatrix.SetColumn(2, new Vector4(0, 0, 1, 0));

    // Going from right to left:
    // convert screen coords to clip coords, then clip coords to world coords
    return vpMatrix.inverse*viewportMatrix;
}

public Vector2 ScreenToWorldPoint(Vector2 point){
    return ScreenToWorldMatrix(camera).MultiplyPoint(point);
}
```

Splitting this conversion into two methods provides flexibility to cache the matrix to convert many points in a single frame.

Custom Projections for the Scene Editor

So with a minimal amount of code, it’s possible to have Unity render a custom projection similar to ones used in classic 2D video games complete with support for perspective, input, and automatic draw order sorting. The last obstacle to tackle is how to get the custom projection working with the Unity scene view for WYSIWYG editing.

Fortunately, recent versions of Unity provide the hooks needed to make this work. First, the camera script will need to have the *[ExecuteInEditMode]* attribute, otherwise it will only work when Unity is in play mode. The following code, builds on the *OnPreCull()* code listed above.

```
private void OnEnable(){
    // Optional, only enable the callbacks when in the editor.
    if(Application.isEditor){
        // These callbacks are invoked for all cameras including
    // the scene view and camera previews.
        Camera.onPreCull += ScenePreCull;
        Camera.onPostRender += ScenePostRender;
    }
}

private void OnDisable(){
    if(Application.isEditor){
        Camera.onPreCull -= ScenePreCull;
        Camera.onPostRender -= ScenePostRender;
    }
}

private void ScenePreCull(Camera cam){
    // If the camera is the scene view camera, call our pre cull method.
    if(cam.cameraType == CameraType.SceneView) OnPreCull();
}

private void ScenePostRender(Camera cam){
    // Unity's gizmos don't like it when you change the worldToCameraMatrix.
    // The workaround is to reset it after rendering.
    if(cam.cameraType == CameraType.SceneView) cam.ResetWorldToCameraMatrix()
}
```

This provides a useful, though imperfect editing experience in the scene view. Though you can edit an object in the inspector just fine, some of the widgets in the scene view are incorrect. In particular, transforms with an offset on the z-axis won't draw at quite the right places, and rect transform handles will be facing on the wrong axis. Some of this can be fixed by disabling the onPostRender event, but it causes different problems. It may not be possible to have a perfect editing experience, though you can still have an improved one.

Conclusion

So with a little bit of extra matrix math, you can save yourself a lot of trouble. Instead of hacking a semi-2D projection on top of a 3D engine, get the engine to do it for you. As a bonus, a lot of other effects such as reflections and shadows are simple variations. In Verdant Skies, we have two extra cameras that render the scene with different projections. Reflections simply point the up vector downwards on the camera, and shadows point it in the direction of the shadows on the ground.

Resources

CustomProjection.cs for Unity

http://www.zeldadungeon.net/2013/11/iwata-asks-a-link-between-worlds-perspective/

Read more about:

Featured Blogs

About the Author

Scott Lembcke

Blogger