Sky's the Limit

VR Gravity-Based Planets

Background

Overview

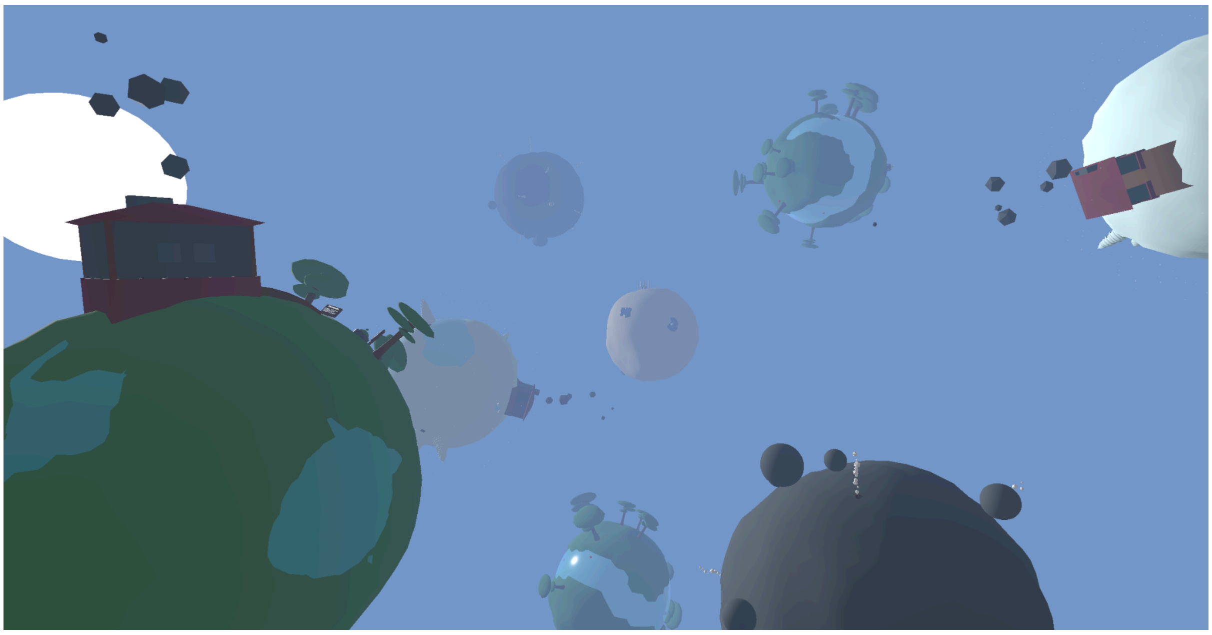


Figure 1. Navigable planets

Sky's the Limit is a VR tech demo showcasing the potential of immersive 3D gravity using VR. The world consists of small, spherical planets with local gravitational fields. The player can find and throw "baby planets" into the sky to create new planets, then use a jetpack to navigate to and explore them. The player can interact with other smaller mechanics surrounding gravity and planets, including geysers and snowballs.

I have long been fascinated with VR's ability to manipulate players' sense of up and down. I was inspired by *Super Mario Galaxy*, *Outer Wilds*, and *PlanetSmith*, as their miniature planets perfectly fit the idea. Unlike those games, this experience would truly immerse the player in the environment by using the player's physical sense of being grounded. Though this would be tricky to implement without nauseating the player, I felt it was worth the challenge.

Aims

For this project, I aimed to develop a nausea-free experience that gave the player the sensation of travelling between different gravitational fields. I wanted players to feel rooted in the planet they stand on while also being able to look into the sky and see a position they had stood at and understood as "the ground" prior. To build on the concept, I planned to add physics interactions that made the most of 3D gravity: rolling boulders and snowballs that could circle the planet, throwable objects that the player could toss into orbit, and means of propulsion so the player could navigate between planets or just see their current planet from a farther distance.

Description and Justification

Tools

I created all models in Blender. I used GitHub for source control, which was invaluable later in development when working around computer-specific issues in the VR lab. I learned that transferring builds via Google Drive made them incompatible with SteamVR's input system, so I used GitHub to ensure the integrity of the build files.

Aesthetic Rationale

Visual aesthetic was a low priority for this project. I opted for a low-poly, block-colour art style to facilitate quick asset creation. I created each planet with a distinct appearance to promote exploration and make each planet exciting. The player's home planet is intentionally barren to promote exploration of other planets.

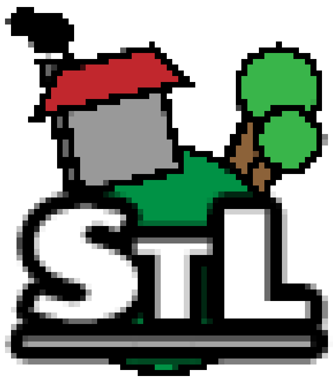


Figure 2. Executable icon

I made the executable icon in Illustrator, representing the starter planet from the point of view of the title screen.

Experience Design

Experience Overview

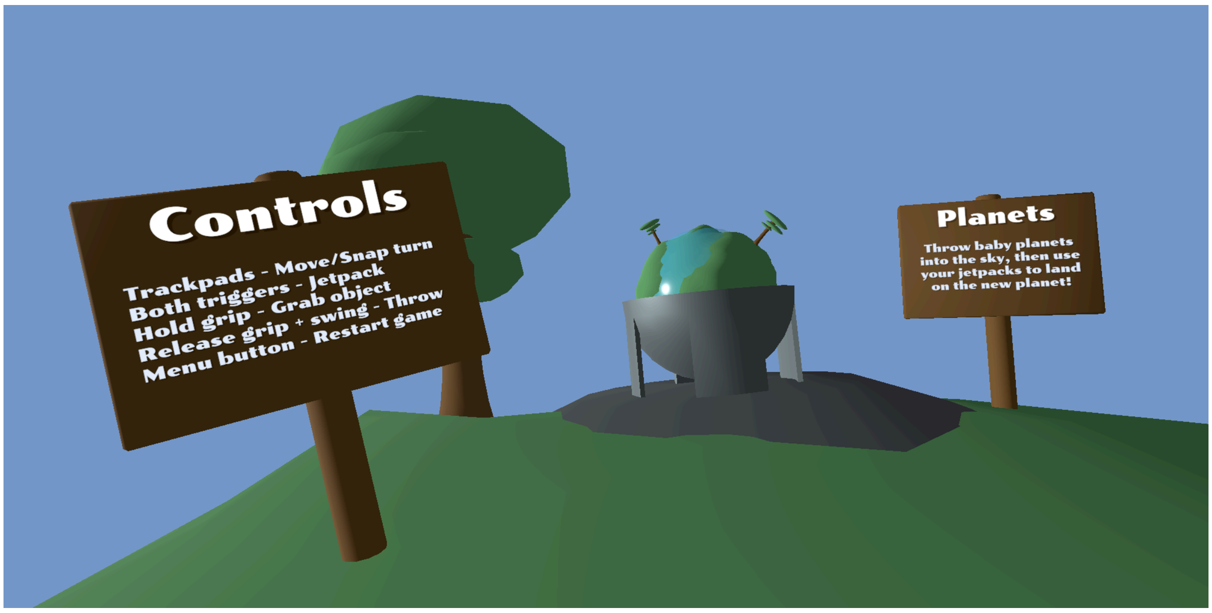

Controls
Trackpads - Move/Snap turn
Both triggers - Jetpack
Hold grip - Grab object
Release grip + swing - Throw
Menu button - Restart game
Planets
Throw baby planets into the sky, then use your jetpacks to land on the new planet!

Figure 3. Home planet

The experience begins on a title screen with a single planet. This displays how empty the system is, providing an initial reference point to compare to later. Players are then taught the controls and how to throw new planets into the sky. After creating a new planet, the player is free to explore and search for more baby planets.

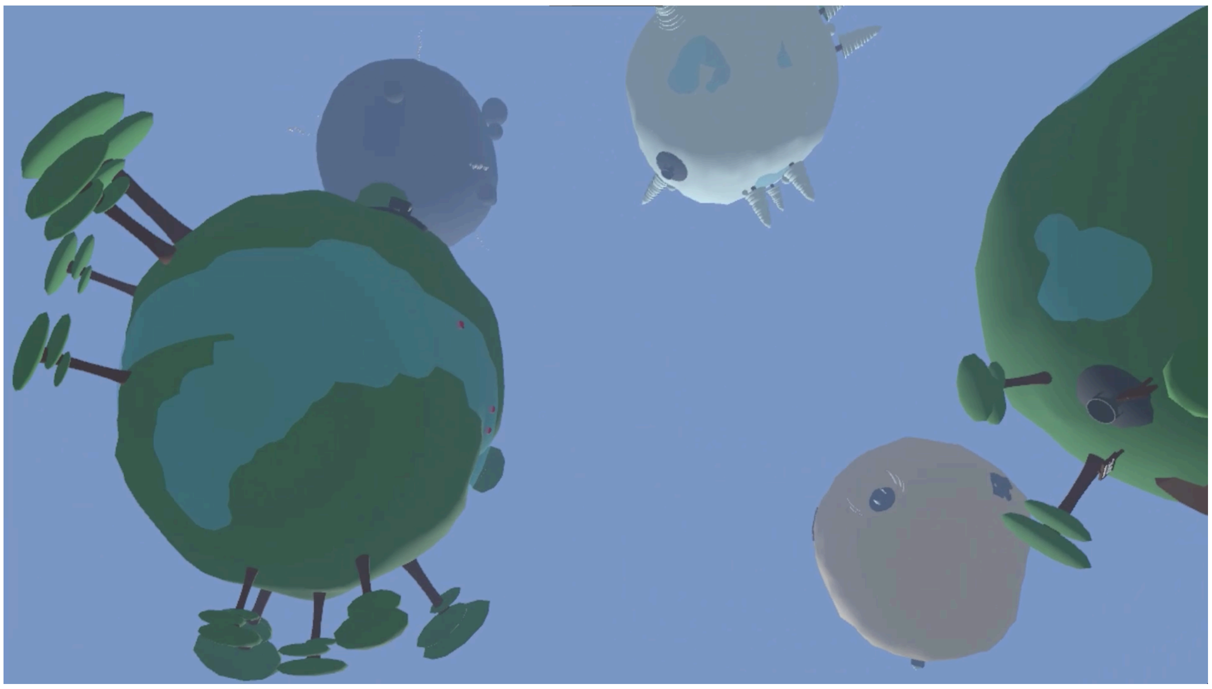


Figure 4. Planets

After creating a handful of planets, players can look into the sky and see how they have filled the space, creating a sense of accomplishment and ownership over the newly created system. Simultaneously, they will see the path they took across planets, recognising locations they visited previously from a new perspective: where they stood when they began the experience may now appear upside down or sideways.

Each planet serves a unique purpose, so players have something to gain from visiting each one. Every planet has at least one baby planet to be found, meaning the player will always be able to create more. While I initially planned to make planets affected by gravity and orbit around the sun, I scrapped this idea to ensure players could easily navigate between planets without having to consider planetary movement.

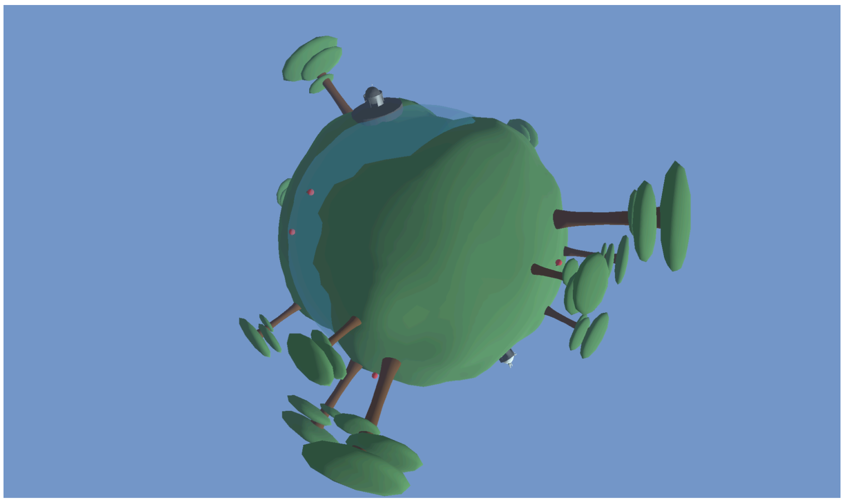


Figure 5. Grass planet

The first, grassy planet's highlight is its apples: players can pick them up, throw them into orbit, and, if skilled enough, catch them as they return from the other side of the planet.

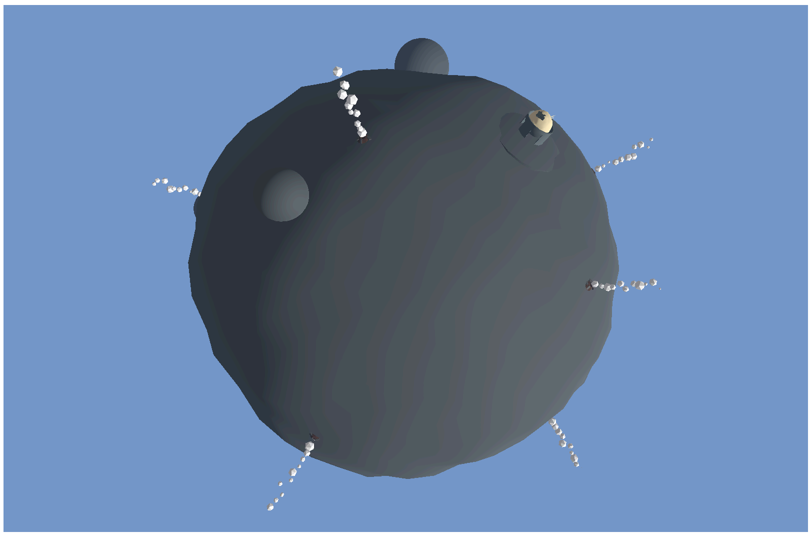


Figure 6. Geyser planet

The geyser planet is a stoney planet with boulders and geysers that boost objects into the air. Boulders often roll into these geysers, showing the player their functionality. If two geyser planets are close together and correctly aligned, boulders can bounce between them.

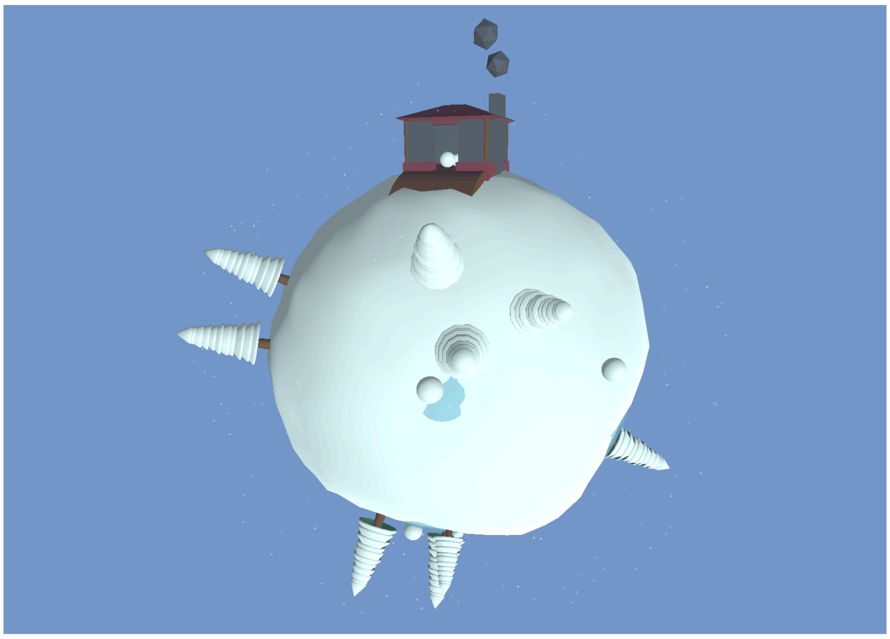


Figure 7. Snow planet

The snow planet features snow particles, snowy trees, ice, a single house, and snowballs. Snowballs can be thrown like apples or rolled across the ground to increase their size.

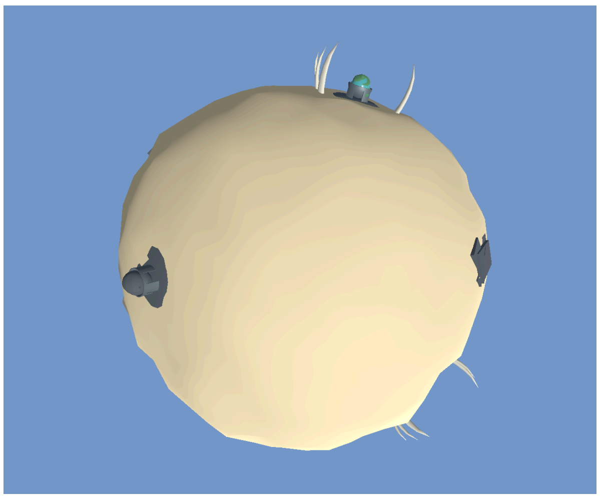


Figure 8. Sand planet

The sand planet is barren but has all three other baby planet types. It serves as a junction point, allowing the player to decide where to go next.

Player and Anti-Nausea

Preventing nausea was one of my highest priorities during development. Without any similar VR planet experiences to use as reference, I extensively tested the player movement and rotation to ensure it would be as nausea-free as possible.

Initially, the player would smoothly turn to stand upright against the current gravity source, including when standing on the planet. This led to a slight smoothing effect on the camera when moving, creating a nauseating effect where the camera would continue to rotate for a moment after stopping movement or changing direction. I solved this by ensuring that grounded objects immediately snap to the desired rotation. This causes a sharp visual cut for the player when landing on a planet, which is relatively insignificant compared to the anti-nausea benefits of preventing the camera rocking effect.

The player uses Rigidbody forces to move. This eases the player into and out of movement to prevent unpredictable, shocking movements. However, without a velocity limit, the character could be sent flying by an unlucky force or move fast enough to be thrown into orbit. I placed a medium-speed velocity cap on the player to ensure that no matter what, they would not be able to move at high speeds. This prevents nausea from fast-paced locomotion, jetpacking, and mass-related physics collisions with boulders and snowballs.

Technical Implementation

SteamVR Issues

Far and away, the most difficult and frustrating area of this project was working with Unity's SteamVR plugin. Using this plugin caused countless issues and greatly slowed development. Its system for inputs via actions and bindings was awful to work with, requiring my personal Steam account to be open to operate and often failing to update when adding new bindings or changing binding type. Thankfully, actions were directly editable via JSON, which made adding new actions possible in these cases. The plugin caused frequent OS crashes while playing in-editor, rendering my personal PC and HMD useless for development. Documentation on the plugin's GitHub was offline during the final days of development, and I noticed many open bug reports when trying to solve issues. Despite constant issues with the SteamVR plugin, I successfully built the project with functional input.

Gravity

Gravity is handled with two scripts: Gravity3D and GravitySource. Gravity3D attaches to Rigidbody objects to override their gravity functionality, while GravitySource enables a gravitational field around an object like a planet.

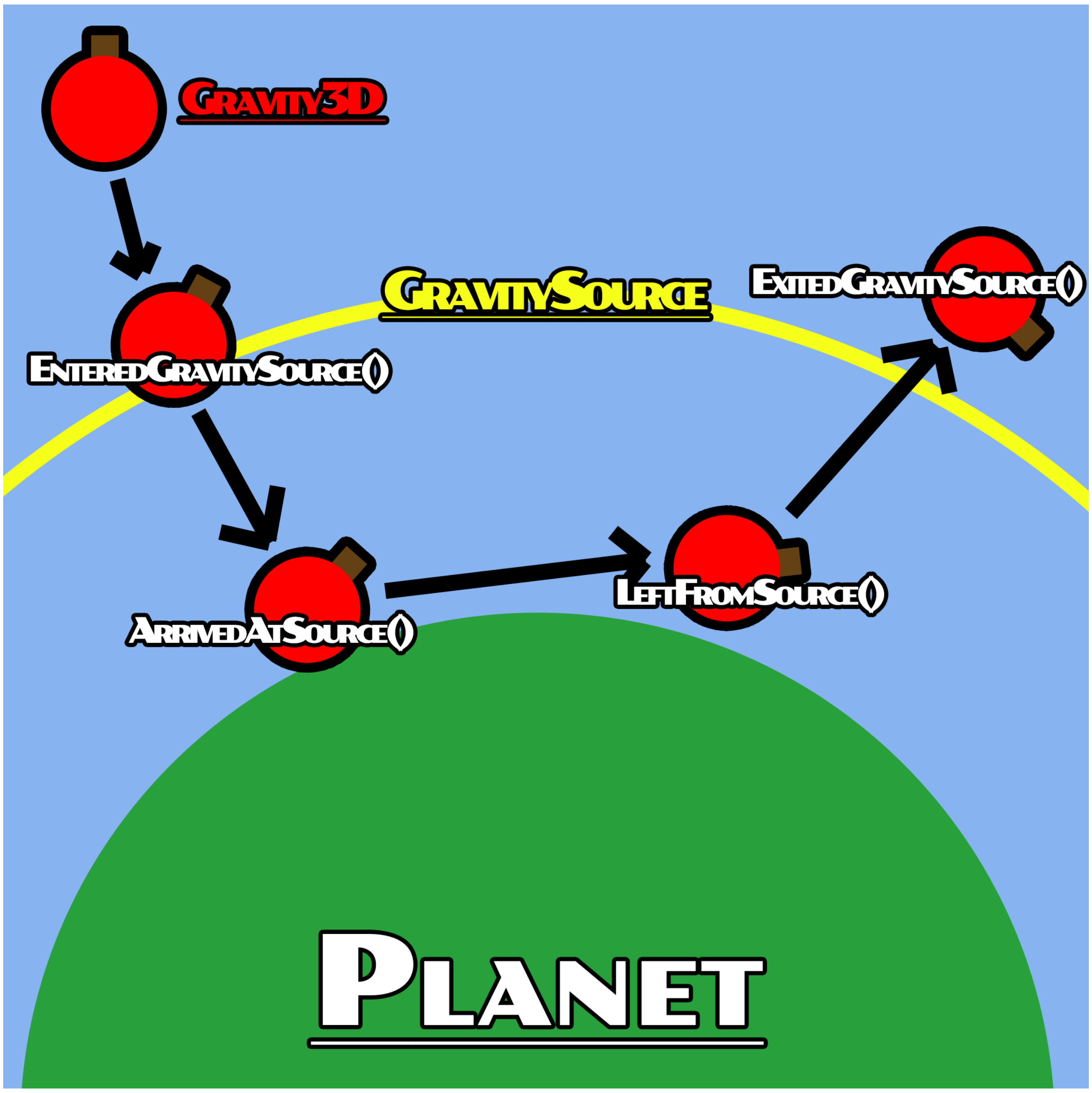
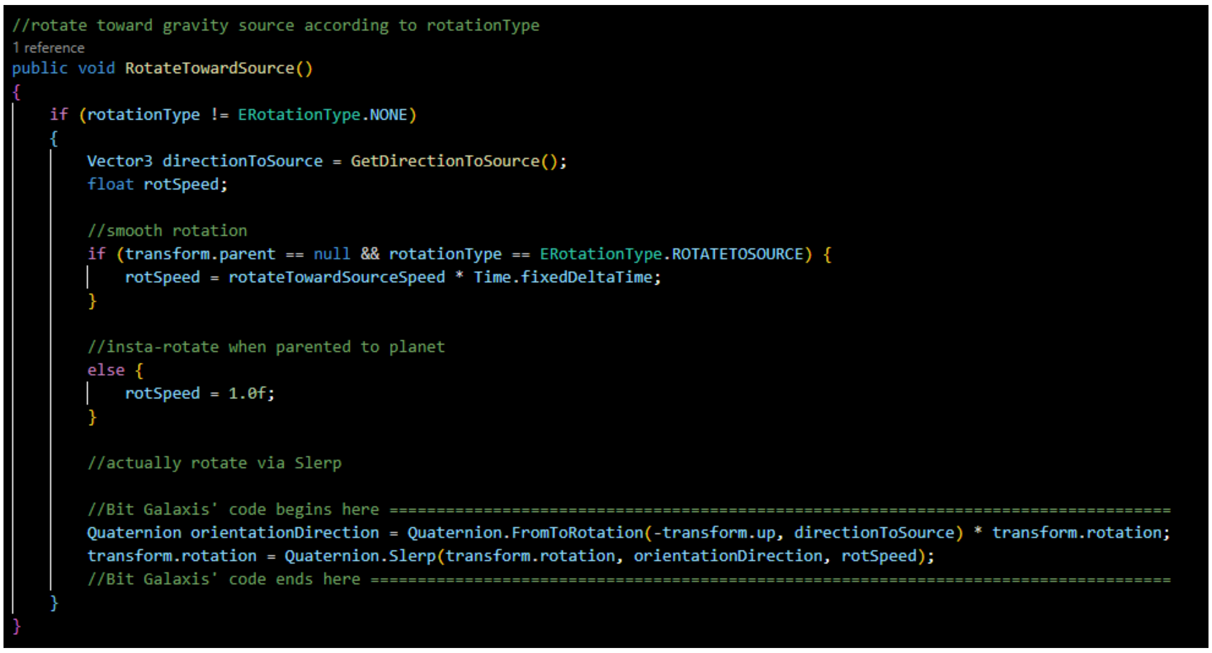


Figure 9. Gravity3D functions and when they are called

Gravity3D overrides the associated Rigidbody's useGravity value, copying and disabling it if enabled. Gravity3D has functions for entering a planet's gravitational field, landing on a planet, leaving a planet, and exiting a planet's gravitational field. When a Gravity3D enters a GravitySource's trigger collider, Gravity3D saves a reference to the new GravitySource and begins applying its gravitational strength each FixedUpdate(). Once the object lands on the planet, it becomes a child of the gravity source. This ensures that any Gravity3D object remains tethered to its planet if the planet moves.

Once no longer grounded on a planet, Gravity3D.standingOnPlanet is set to false. This allows other scripts to access the object's grounded state; the player uses this for movement. The object remains as a child of the GravitySource until leaving its gravitational field trigger collider. At this point, it clears its parent but still uses the planet as its source of gravity. Although unrealistic, this means objects that leave a planet's gravity will not float away into space but will instead orbit the planet or eventually return to its surface. This was primarily designed to ensure the player always has a home planet to return to should they fall; their previous planet will always be the "ground" and provides a consistent sense of up and down until they enter another planet's gravitational field.



```
//rotate toward gravity source according to rotationType
1 reference
public void RotateTowardSource()
{
    if (rotationType != ERotationType.NONE)
    {
        Vector3 directionToSource = GetDirectionToSource();
        float rotSpeed;

        //smooth rotation
        if (transform.parent == null && rotationType == ERotationType.ROTATETOSOURCE) {
            rotSpeed = rotateTowardSourceSpeed * Time.fixedDeltaTime;
        }

        //insta-rotate when parented to planet
        else {
            rotSpeed = 1.0f;
        }

        //actually rotate via Slerp

        //Bit Galaxis' code begins here ==================================================================================
        Quaternion orientationDirection = Quaternion.FromToRotation(-transform.up, directionToSource) * transform.rotation;
        transform.rotation = Quaternion.Slerp(transform.rotation, orientationDirection, rotSpeed);
        //Bit Galaxis' code ends here ====================================================================================
    }
}
```

Figure 10. Gravity3D.RotateTowardSource()

Objects inside a gravity source may rotate toward the source depending on their rotationType. ERotationType includes three options for gravity-affected Rigidbodies: NONE, where the object will maintain its original rotation; ROTATETOSOURCE, where the object will smoothly rotate toward the source of gravity at rotateTowardSourceSpeed speed; and SNAPTOSOURCE, where the object will instantly snap to the correct rotation without smoothly transitioning. SNAPTOSOURCE is used by the player, usually a ROTATETOSOURCE object, to ensure they are properly upright at all times once they have landed on a planet. This method prevents the nauseating camera rocking effect described earlier. When the player enters a new gravitational field, they spherically interpolate toward the correct rotation. Although I initially expected this to be nauseating, it works quite well to smoothly transition the player between their old and new “up and down” without disorienting them.

Player

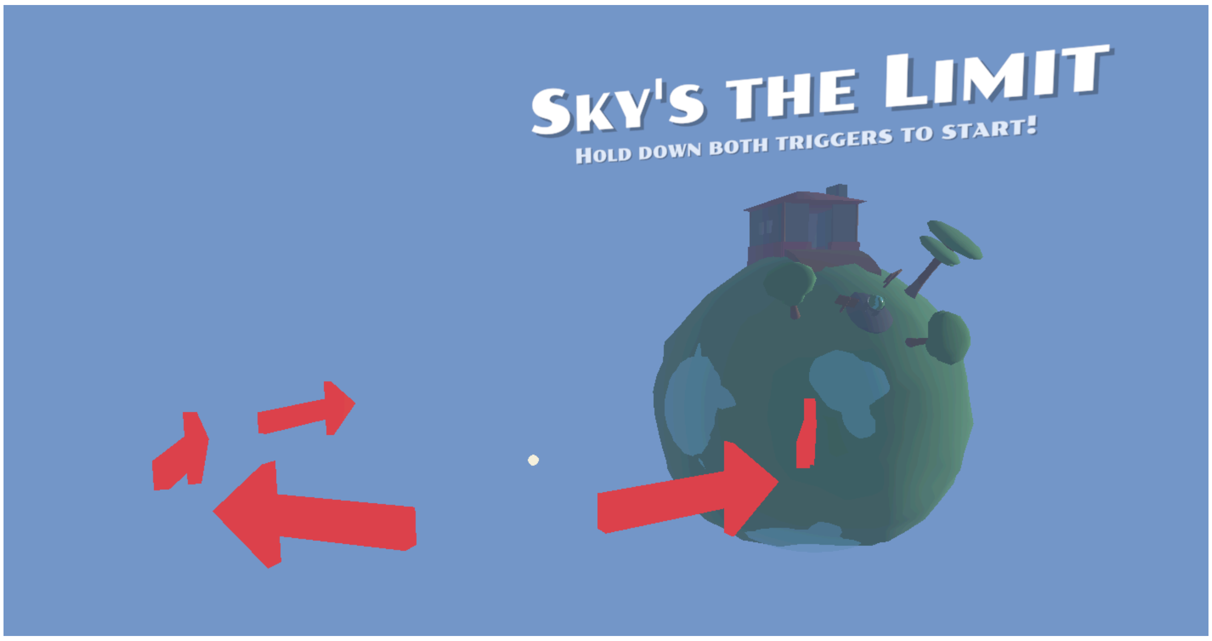
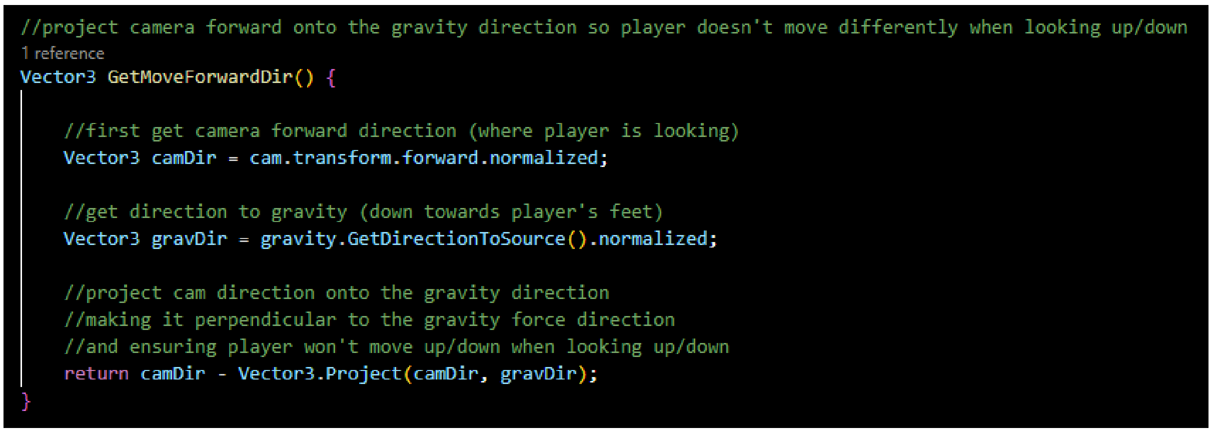


Figure 11. Arrows on the title screen correct mis-rotated players

I initially built on the SteamVR default player for my player controller but later switched to building on SteamVR's XR Rig for greater flexibility. SteamVR's player included long, difficult-to-read scripts, loads of bloat, and many tracking and movement issues; development was much smoother after abandoning it. Even after building a new character, some SteamVR tracking issues persisted, such as issues with rotation and initial player position. My workaround for these issues was to use arrows on the title screen to point players toward the correct direction to look, then to disable the title screen player and enable a separate player, which would always begin in the same position and rotation.

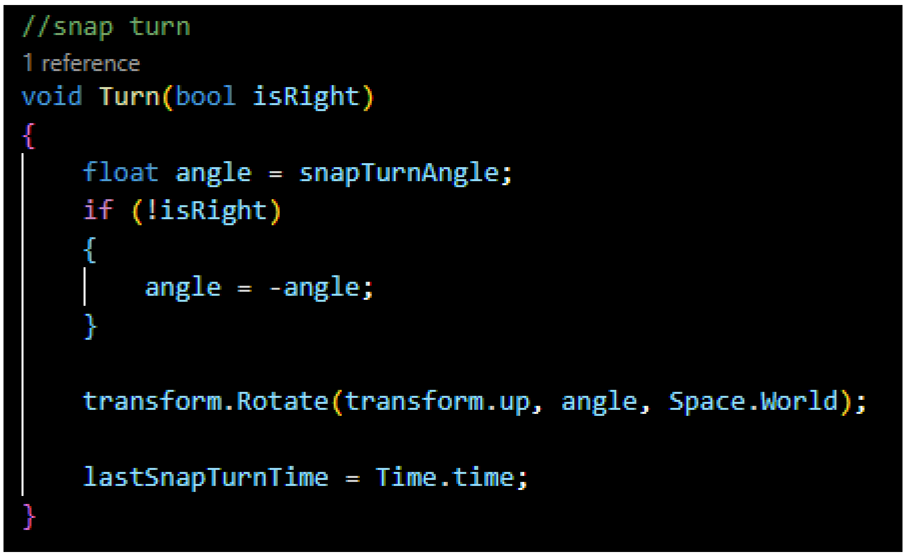
By default, XR Rig only includes a position-tracked camera, making it much easier to work with. I added tracked hands using SteamVR's Tracked Object and Behaviour_Pose components, which thankfully worked without any of the bloat from the original player hands. I then built locomotion and grabbing/throwing interactions with my own scripts: PlayerMvmt and MyHand.



```
//project camera forward onto the gravity direction so player doesn't move differently when looking up/down
1 reference
Vector3 GetMoveForwardDir() {

    //first get camera forward direction (where player is looking)
    Vector3 camDir = cam.transform.forward.normalized;

    //get direction to gravity (down towards player's feet)
    Vector3 gravDir = gravity.GetDirectionToSource().normalized;

    //project cam direction onto the gravity direction
    //making it perpendicular to the gravity force direction
    //and ensuring player won't move up/down when looking up/down
    return camDir - Vector3.Project(camDir, gravDir);
}
```

Figure 12. PlayerMvmt.GetMoveForwardDir()

I implemented locomotion movement for a more immersive experience than teleport movement; smooth trackpad movement allows players to watch planets in the sky pass by as they travel around a planet, better communicating a sense of scale and the manipulation of up and down. Locomotion works as it would in any first-person controller with a couple of complications: movement had to be relative to the VR camera rotation as well as the player's alignment to their current gravity source, and should not be affected by whether the player was looking relatively up or down. I accomplished this by getting the camera's forward vector, creating a second vector by projecting the camera's forward vector onto the directional vector from the player to the gravity source, and then subtracting the second vector from the camera's forward vector. This creates a vector that is always perpendicular to what would be the down vector of the player, meaning the player will move relative to their camera rotation and not relative to their camera's local up or down rotation.



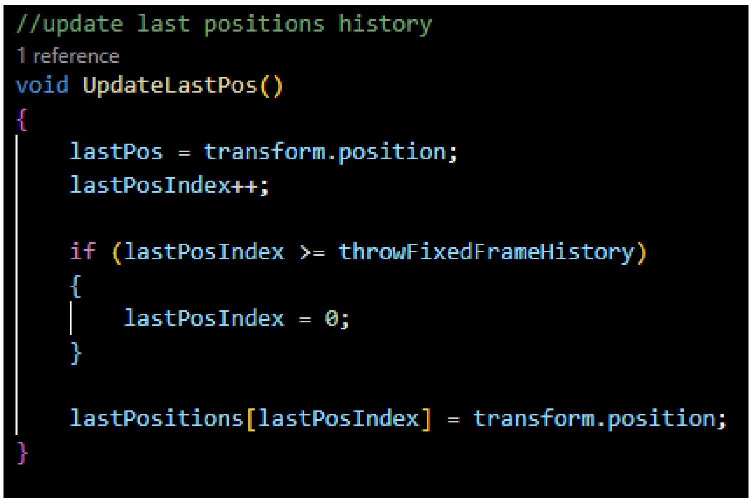
```
//snap turn
1 reference
void Turn(bool isRight)
{
    float angle = snapTurnAngle;
    if (!isRight)
    {
        angle = -angle;
    }

    transform.Rotate(transform.up, angle, Space.World);

    lastSnapTurnTime = Time.time;
}
```

Figure 13. PlayerMvmt.Turn()

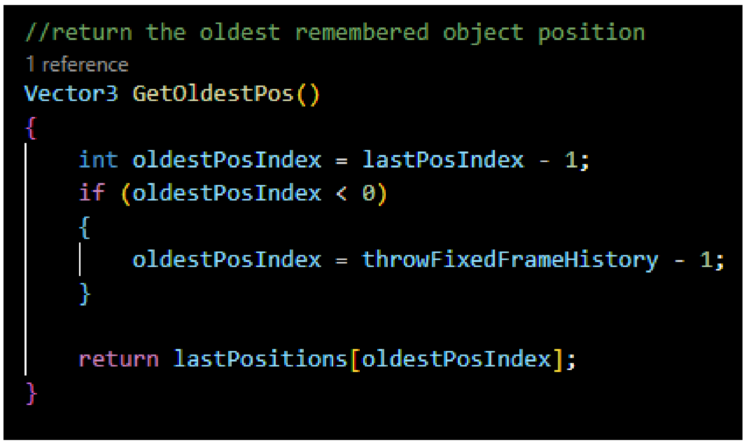
I initially based my snap turning solution on the default SteamVR player’s implementation, but changed it to work better with 3D gravity. This is relatively simple, rotating the player around their up vector in world space. I implemented the SteamVR player’s limit on snap turning as I found it to be necessary with the Vive controllers’ high touch sensitivity.

Grabbables and Throwing

The player can pick up and throw grabbables, which include apples, snowballs, and baby planets. GrabbableObject attaches to a Rigidbody, allowing it to be picked up by the player. When an empty hand overlaps an unheld grabbable, it saves a reference to it. If the player then presses grip, MyHand sends IInteractable.SetInteract(true) (an interface call) to the grabbable. The grabbable then disables its collision and sets its Rigidbody to kinematic, saving its previous kinematic state to be restored when the object is dropped. The object is then attached to the hand. Dropping the object unparents it from the hand, restores its previous kinematic state, and prepares the hand to pick up another object. Dropped objects momentarily keep their collision disabled to prevent collisions with the hand, though this could cause issues such as throwing an object through a wall or dropping it through the floor.



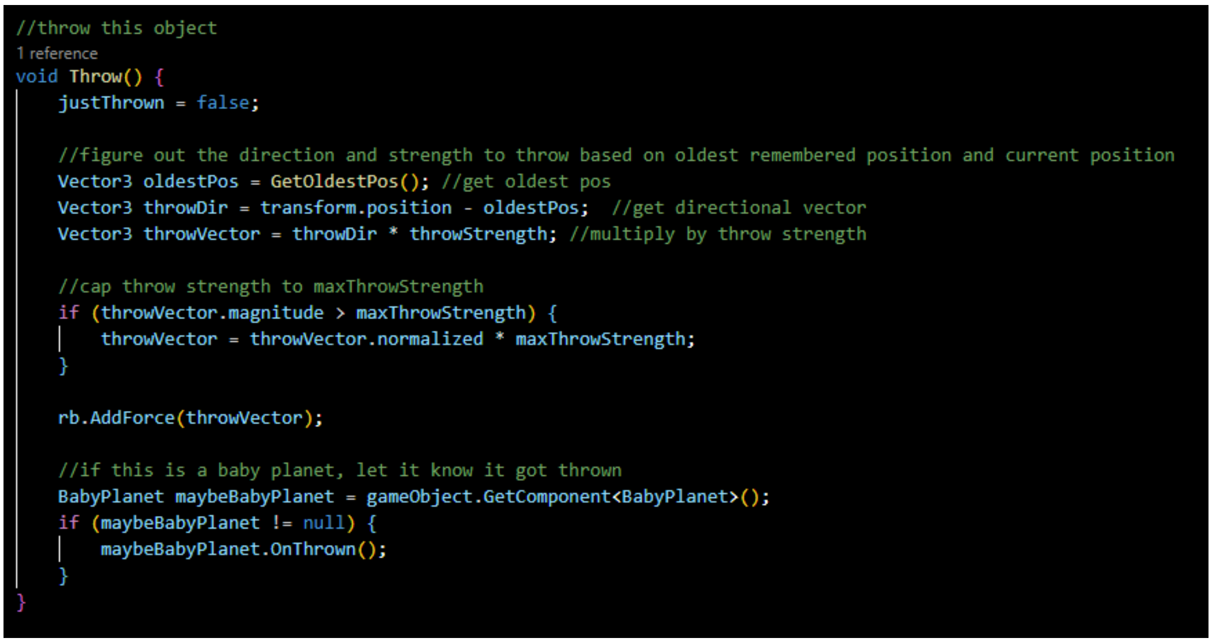
```
//update last positions history
1 reference
void UpdateLastPos()
{
    lastPos = transform.position;
    lastPosIndex++;

    if (lastPosIndex >= throwFixedFrameHistory)
    {
        lastPosIndex = 0;
    }

    lastPositions[lastPosIndex] = transform.position;
}
```



```
//return the oldest remembered object position
1 reference
Vector3 GetOldestPos()
{
    int oldestPosIndex = lastPosIndex - 1;
    if (oldestPosIndex < 0)
    {
        oldestPosIndex = throwFixedFrameHistory - 1;
    }

    return lastPositions[oldestPosIndex];
}
```

Figure 14. GrabbableObject.UpdateLastPos(), Figure 15. GrabbableObject.GetOldestPos()

When dropped, grabbable objects with GrabbableObject.isThrowable set to true will be "thrown" rather than dropped. If the player has not significantly moved the object in the last few frames, this will appear the same as dropping it.



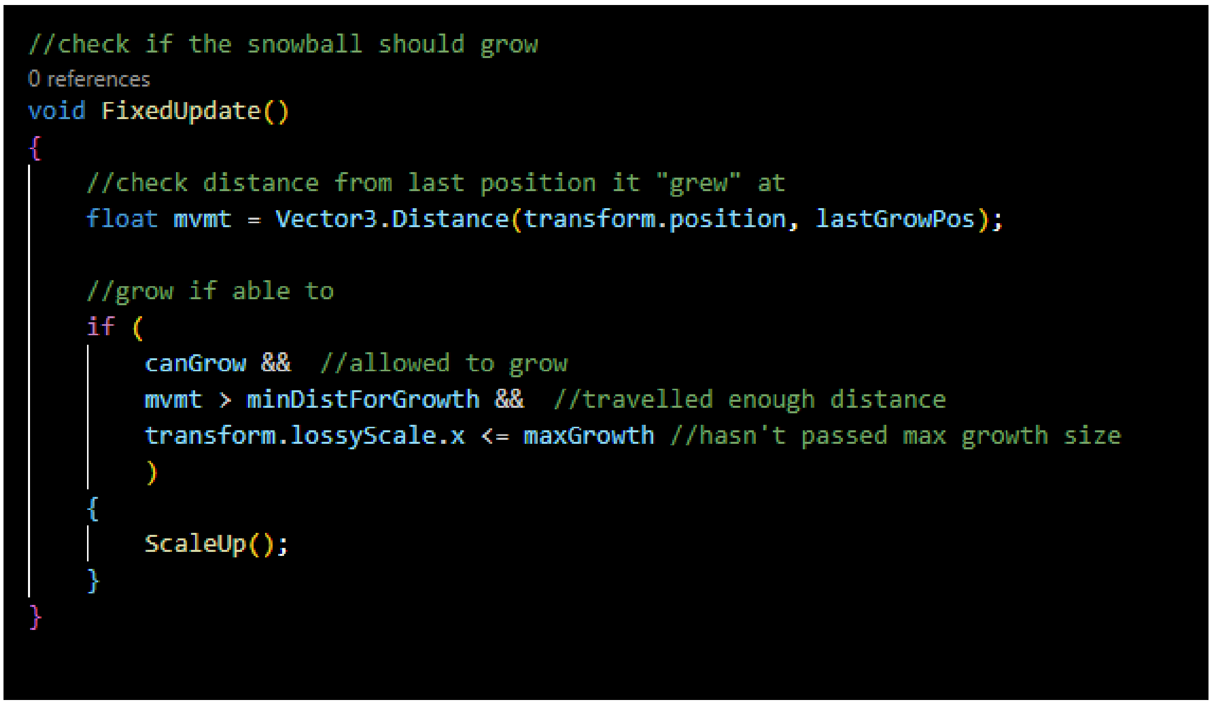
```
//throw this object
1 reference
void Throw() {
    justThrown = false;

    //figure out the direction and strength to throw based on oldest remembered position and current position
    Vector3 oldestPos = GetOldestPos(); //get oldest pos
    Vector3 throwDir = transform.position - oldestPos;  //get directional vector
    Vector3 throwVector = throwDir * throwStrength; //multiply by throw strength

    //cap throw strength to maxThrowStrength
    if (throwVector.magnitude > maxThrowStrength) {
        throwVector = throwVector.normalized * maxThrowStrength;
    }

    rb.AddForce(throwVector);

    //if this is a baby planet, let it know it got thrown
    BabyPlanet maybeBabyPlanet = gameObject.GetComponent<BabyPlanet>();
    if (maybeBabyPlanet != null) {
        maybeBabyPlanet.OnThrown();
    }
}
```

Figure 16. GrabbableObject.Throw()

Grabbables keep a history of their positions in the last five FixedUpdate()s. When thrown, a grabbable finds the distance between its oldest remembered position and its current position, then multiplies that distance by throwStrength. This vector's magnitude is capped at maxThrowStrength to prevent physics bugs from flinging objects ridiculously far. The final vector is then applied as a force to the object, creating a smooth throw from the player's hand. This solution ensures consistent throws and prevents the issues of using a single-frame history.

Other Interactables

Geysers are very simple: they track all objects inside their trigger collider and apply a force to them on FixedUpdate(). The force applied to players is much less than other objects to prevent nausea.



```
//check if the snowball should grow
0 references
void FixedUpdate()
{
    //check distance from last position it "grew" at
    float mvmt = Vector3.Distance(transform.position, lastGrowPos);

    //grow if able to
    if (
        canGrow &&  //allowed to grow
        mvmt > minDistForGrowth &&  //travelled enough distance
        transform.lossyScale.x <= maxGrowth //hasn't passed max growth size
        )
    {
        ScaleUp();
    }
}
```

Figure 17. Snowball.FixedUpdate()

Snowballs were unexpectedly complex to implement. Through trial and error, I landed on a solution where snowballs track their distance from their last grounded position or the last position they grew at, then scale up a small amount once beyond that distance. I implemented a short timer after growing before tracking position again to prevent the object from scaling up infinitely due to the movement from scaling into the ground and being moved by Unity's collision detection. They also stop growing once around the player's size to prevent infinite growth.

Reflection

I am content with this project’s outcome as a demonstration of a mechanic that could be built into a full game. I am very pleased with the efficacy of anti-nausea measures in the player’s movement; while I recognise this mechanic will always be prone to causing nausea for some players, I believe I did well in making the player feel grounded on planets and not like a rolling physics object. I am confident this mechanic could be built beyond a tech demo into a full-fledged VR game.

It is very unfortunate that the SteamVR plugin was required for this project. It plagued the project with many undue issues and is clearly outdated. In the future, I could circumvent tracking and rotation issues by starting from scratch without the SteamVR plugin and opting for a more recent version of Unity with a different VR workflow. While I would like to build for a non-wired platform like Meta Quest 2 to improve player mobility, the planet-throwing mechanic would be tricky to implement as it cannot rely on baked lighting and would, therefore, be too computationally expensive.

I could build on the project’s content and gameplay in many ways. Firstly, I would add more physics-based interactions; I had planned to include physics interactables such as glue that could join two physics objects (inspired by *The Legend of Zelda: Tears of the Kingdom*’s building mechanics), rockets, magnets and a magnetised planet, and conveyor belts. Other mechanics could relate more to the planetary theme, such as scaling the player to a miniature or gargantuan size or a weather control device that could raise the water level with rain or melt frozen walls with the sun. These mechanics would be used in puzzles found on the planets and would reward the player with a new baby planet upon completion.

I would also like to build on the visual aesthetic with lusher green planets, underwater coral, and physics features like willow trees with hanging leaves and vines that respect the local gravitational field. The project would greatly benefit from audio, including interaction sound effects and a unique soundtrack as the player lands on each type of planet.

Word count: 2722

References

Annapurna Interactive. (2019). *Outer Wilds* [Video game].

Incandescent Games. (n.d.). *PlanetSmith* [Video game].

Nintendo. (2007). *Super Mario Galaxy* [Video game].

Nintendo. (2023). *The Legend of Zelda: Tears of the Kingdom* [Video game].