VECTOR VR - Design Document

[redacted] | SPH4UR Summative Project 2026

---

1. Introduction

What is VECTOR VR?

VECTOR is a virtual reality application that helps students learn physics by visualizing motion and forces in 3D space. Instead of reading about velocity vectors in a textbook, students can throw a ball and see the velocity arrows appear in real-time around it.

Why This Project Matters

Many students don't have access to physics labs or expensive equipment. VECTOR VR brings the lab experience into any space with a VR headset (that a library or household may have), making physics more accessible and engaging.

Project Goals

- Create a working VR physics sandbox
- Display velocity and force vectors visually in 3D
- Allow time manipulation (slow motion and pause)
- Learn VR development with Godot Engine

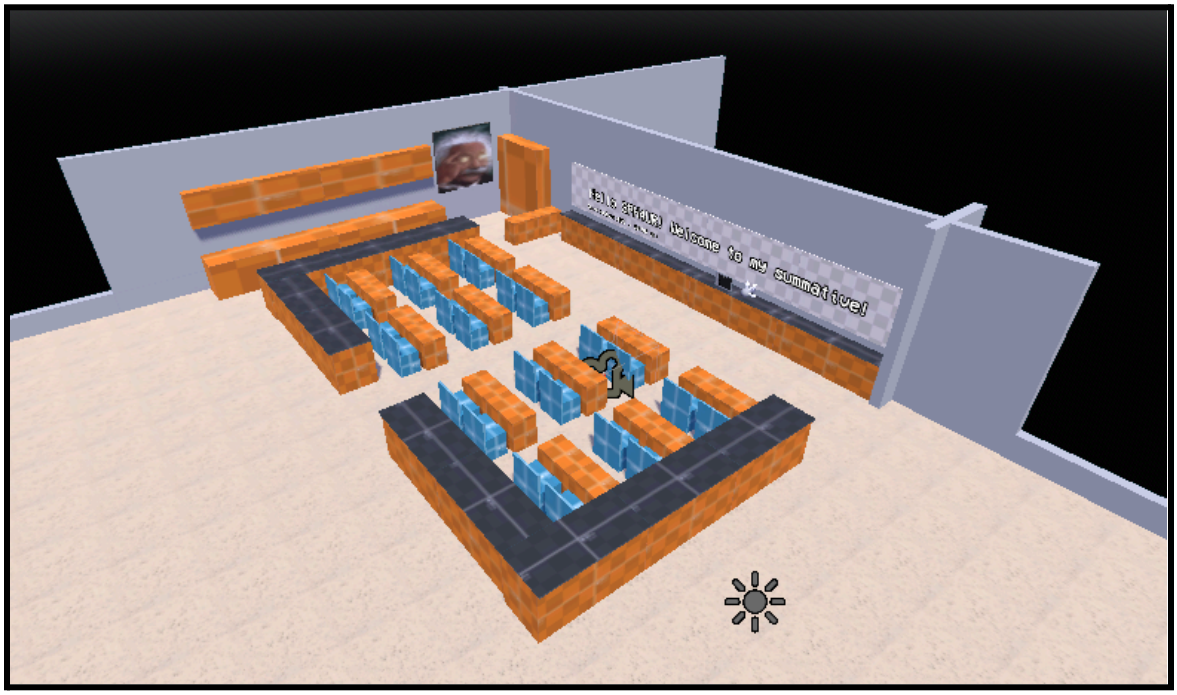
---

2. How It Works (System Architecture)

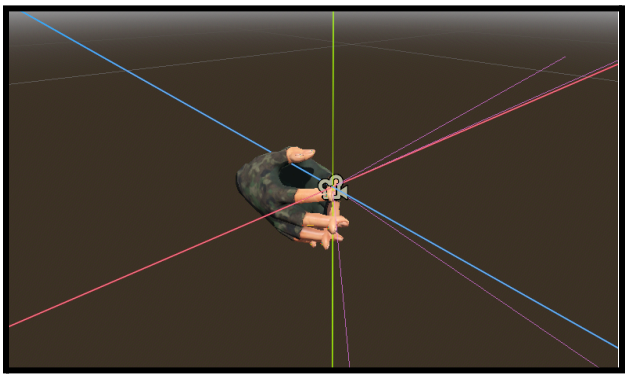
The Big Picture

VECTOR VR consists of four main parts working together:

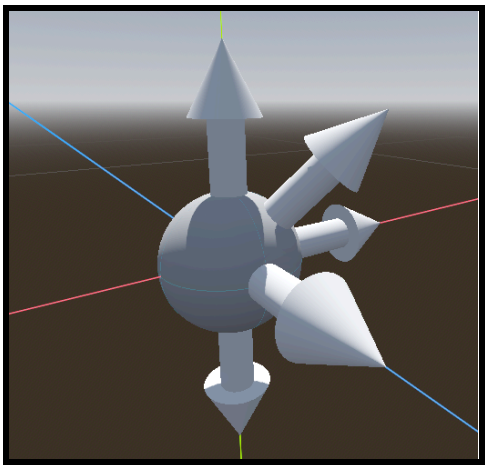
1. **The Virtual Lab** - A 3D classroom environment where physics happens; this includes:
   - **meshes** (visible representations of space)
   - **congruent collision shapes** (for object collision)

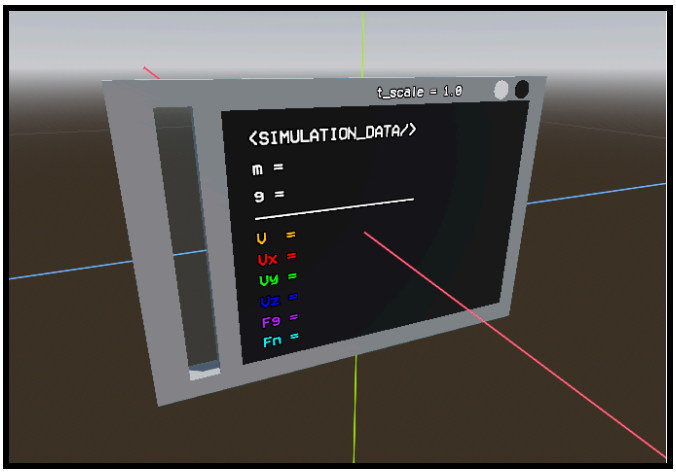


2. **The User** - You, wearing a VR headset with hand controllers.



3. **The Physics Ball** - An interactive object that demonstrates motion.





4. **The Data Tablet** - A virtual screen showing real-time measurements, as well as environment variables.

Technology Stack (Tech used for the project)

- **Godot (4.5)** - Game engine that runs the simulation.
- **OpenXR** - Standard library that makes VR headsets work.
- **GDScript** - Programming language (similar to Python).
- **Meta Quest 2** - VR headset used for testing.
- **USB-A → USB-C cable** - Used for android debugging.

---

3. The Physics Behind It

What Physics Concepts Are Shown?

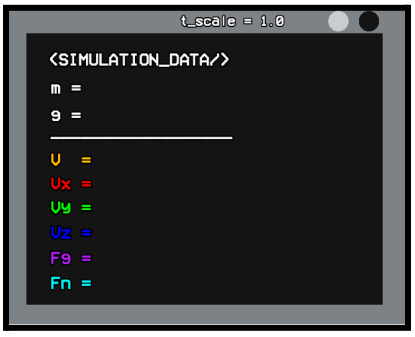
Velocity Vectors

- Arrows showing how fast and in what direction the ball moves.
- **Four vectors displayed:**
  - **Red arrow** (X-axis) - left/right movement
  - **Green arrow** (Y-axis) - up/down movement
  - **Blue arrow** (Z-axis) - forward/back movement
  - **Orange arrow** - combined "resultant" velocity

Force Vectors

- **Purple arrow (Gravity)** - Always points down (unlike reality that points towards the center of the earth)
- **Cyan arrow (Normal Force)** - Points up from surfaces when ball touches them

Measurements Displayed (on the tablet)



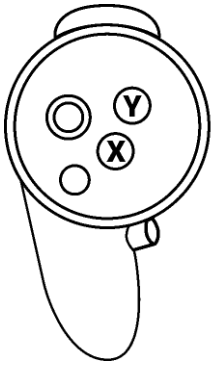
- *Mass (m)
- *Gravity (g)
- *Time Scale (t_scale)
- Velocities (V, Vx, Vy, Vz)
- Forces (Fg, Fn)

** Certain variables can be manually changed, currently there isn't logic in place to do so through the experience itself.*

---

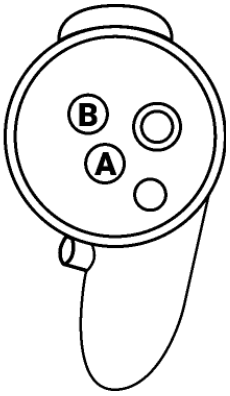
4. User Experience Design

Controls



Left Controller:

- **Thumbstick**: Walk around the lab.
- **Trigger**: Activate slow motion.
- **Y Button**: Teleport tablet to hand.
- **Grip**: Pick up objects.



Right Controller:

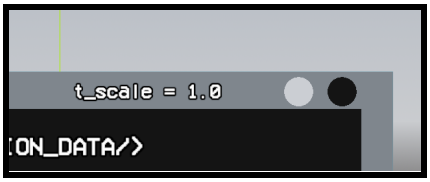
- **Thumbstick**: Turn left/right.
- **Trigger**: Pause/unpause time.
- **B Button**: Teleport ball to hand.
- **Grip**: Pick up objects.

The Virtual Lab

- Desks and lab benches arranged like the real classroom.
- Whiteboards with a welcome message.
- Open floor space for throwing the ball.
- There is a red structure, used mainly to demonstrate normal force being perpendicular to the surface.

5. Key Features & How They Work

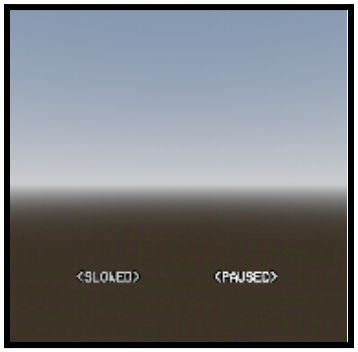
Time Manipulation



Slow Motion (Left Trigger)

- Sets the physics engine to 0.2x of normal speed
- User movement speed adjusts so it feels normal.
- Cyan light on the tablet turns on.
- "<SLOWED>" label appears in the headset as a heads up display (HUD).

Pause Time (Right Trigger)



- Freezes the ball completely in mid-air.
- The ball's vectors remain for display.
- Orange light on the tablet turns on.
- "<PAUSED>" label appears in headset

Why This Is Tricky: Pausing isn't just setting physics speed to zero — that would prevent user movement as well. Instead, when pausing, the code:

1. Saves current velocity and position.
2. Turns off gravity and freezes the ball.
3. When unpaused, it restores everything exactly.

Vector Visualization

How Vectors Update:

- Every frame (60 times per second), code **reads** the ball's velocity (reading the physics engine's values).
- Arrows smoothly grow/shrink to match **magnitude**.
- Arrows rotate to point in movement **direction**.
- Linear interpolation (lerp) is used to **soften** the changes by applying them over a small delay (*).

Normal Force Detection: Uses "raycasting" — imagine a laser beam shooting down from ball:

- If it hits a surface → calculate and show normal force.
- If in mid-air → hide the normal force vector.
- Updates constantly as the ball rolls on surfaces.

---

6. Technical Implementation

Code Structure

Note: These are code files; ".gd" means this is a file with code from the Godot language — this is what is used in the engine. It's similar to the Python programming language.

Main.gd ("Control Center")

- Initializes VR interface.
- Connects all parts together.
- Handles time pause logic.
- Manages ball/tablet teleportation.

xr_control_logic.gd (User Input)

- Reads controller buttons.
- Handles walking and turning.
- Triggers time manipulation.
- Sends signals to other scripts based on user action.

projectile.gd (Ball + Vectors)

- Calculates vector sizes.
- Updates arrow positions and rotations.
- Detects surface contact for normal force.

tablet.gd (Data Display)

- Updates all text labels.
- Manages indicator lights.
- Stores frozen values during pause.
- Formats numbers for readability.

Scene/Object Hierarchy

Main

├── User (VR player)

│ ├── Camera (eyes)

│ ├── Left Controller (hand)

│ └── Right Controller (hand)

├── World (the lab environment)

├── Projectile (the physics ball)

│ ├── Ball (RigidBody3D - actual physics object)

│ └── Vectors (visual arrows)

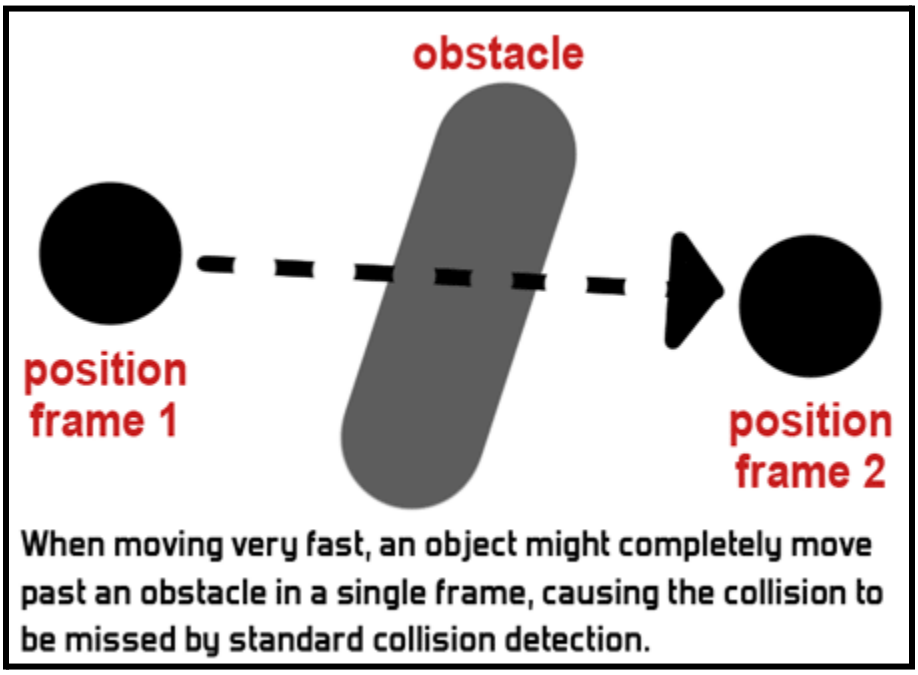
└── Tablet (data display)

---

7. interesting bugs in the program!

Bug 1: Ball Clipping Through Floor (or any surface)

Problem: At high speeds, the ball may go through surfaces.
The Physics:



For object collision to be registered, two collision areas must overlap. Usually this is no issue. However, if an object is moving fast enough, it can bypass collision entirely. But why is this?

For each frame, the order of processing is (simplified):

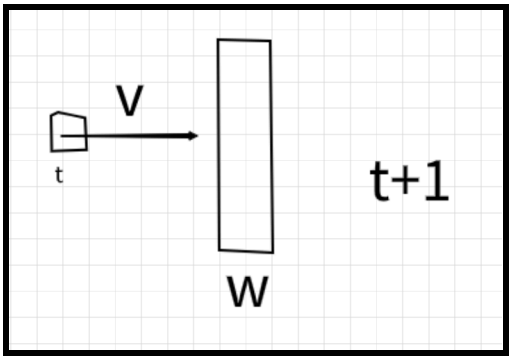
1. User input
2. Physics
    a. Movement
    b. Collision
3. Graphics

This order matters a lot — essentially before checking any collision or things that cannot happen (i.e. overlap of objects in the same space), the engine "entertains" it. Only after processing all the movement does it go back and make sure it's possible or not possible.

I did the math, there is an equation to determine whether an object will clip (that is the widely-used term for this occurence) through a surface. To get it we must solve an inequality

Consider the following variables:
(*w*) represents the thickness of a surface (m)
(*v*) represents the NORMAL velocity INTO the surface (m/s)
(d) represents the distance travelled in a frame
(t) represents the time occurred between frames (s)
(f) represents the tick (refresh) rate (Hz)

In order to clip through a surface, the distance travelled <u>in 1 frame</u> must be greater than the width of the surface.



⇒ $d > w$

Distance is velocity multiplied by time:

⇒ $vt > w$

isolate velocity (as it will become the variable we can solve for to get the MINIMUM velocity to clip)

⇒ $v > w/t$

$\Rightarrow v > wf \mid (because\ f = 1/t)$

This means that for any given width and refresh rate, the minimum velocity can be calculated. Take VectorVR as an example.

The floor's width (w) is **0.25m**.

The engine's tickrate (f) is **60Hz**.

Therefore, using $v > wf$

$\Rightarrow v > (0.25m)(60Hz)$
$\Rightarrow$ **v > 15m/s** or about only **54km/h**

However, because a person realistically can't throw a ball that fast, it's not something to be too worried about. That would be the case, except in tests I was able to throw the ball high enough such that when it got to the ground it accelerated to clip speed (in very niche cases).

Although this is an extremely simple fix, I won't be making it because it's a tribute to this learning moment I had — might help someone learn something new.

Bug 2: Vectors Jittering

Problem: Arrows update too fast, causing shaky visuals.
Solution: Linear interpolation smooths changes over time. (This does cause another issue: immediately inaccurate vectors.)

Challenge 3: Time Pause Bugs

Problem: Ball drifted or fell when unpausing. (A multitude of reasons for this, not gonna get into it).
Solution: Store exact position, disable gravity during pause, restore everything precisely

Challenge 4: Normal Force Calculation

Problem: Hard to detect when the ball touches surfaces.
Solution: Raycast detection checks for contact 60 times/second.

---

8. Future Improvements

Features to Add

- Variable environmental factors (mass, gravity)
- Friction visualization
- Projectile motion presets (e.g., perfect parabola)
- More things to actually do; energy data, trajectory drawing, etc.

Technical Enhancements

- Tutorial system for first-time users.
- Save/load simulation states.

---

10. Conclusion

VECTOR VR successfully demonstrates key physics concepts through immersive visualization. By combining accurate physics simulation with intuitive controls and time manipulation, it creates a learning tool that makes abstract concepts tangible.

The project taught valuable lessons in VR development, physics engine integration, and user experience design. Most importantly, it proves that complex physics doesn't need expensive lab equipment - just a headset and curiosity.
