### **The Initial Setup Prompt**

Act as an expert C# and MonoGame developer specializing in 3D graphics and physics integration using the BEPUphysics v2 library. I am providing you with the complete codebase for my 2D CRPG project, "Project Vagabond." My goal is to have you guide me through a multi-step process of integrating a high-quality, 3D physics-based dice rolling system.

I will provide you with a series of detailed, sequential prompts. Each prompt will describe a specific task, such as creating a new class or modifying an existing one. It is critical that you follow these rules for every response you provide throughout this entire process:

1. **Full Code Output:** For each prompt, you must output the complete, full code for the requested file(s).
2. **No Brevity:** You must not use brevity comments like // ... or // rest of code is unchanged. The entire file content must be generated every time.
3. **High Quality:** Your output should be fully functional, well-structured, and adhere to best practices for C# and MonoGame development.
4. **Patience:** You will wait for my next prompt before proceeding to the next step. Do not try to anticipate or combine steps.

Your first task is to analyze the provided codebase to understand its structure. Pay close attention to the ServiceLocator, the Global settings class, the main game class Core.cs, and the primary game scene TerminalMapScene.cs, as these will be key integration points.

Once you have analyzed the code and are ready to begin, you must respond with the following exact phrase and nothing else:

**"Analysis complete. I understand the project structure and am ready for the first step."**

Do not provide any code or further explanation until I give you the first official step prompt.

#### **Step 1: The Physics Foundation**

**Prompt:**

Act as an expert C# and MonoGame developer specializing in 3D graphics and physics integration using the BEPUphysics v2 library. Your task is to create the foundational physics simulation class for a 3D dice rolling system.

Create a new C# file named PhysicsWorld.cs. This class will encapsulate the entire BEPUphysics simulation. It must be responsible for creating and managing the physics world, including a floor and invisible walls to contain the dice. The dimensions of this container should be based on the game's virtual screen resolution, which you can access from a static Global class.

The class must include:

1. A constructor that initializes the BEPUphysics Simulation, BufferPool, and SimpleThreadDispatcher.
2. A method to create the static environment: a flat, non-moving plane for the floor and four static walls around the perimeter of the virtual screen (Global.VIRTUAL\_WIDTH, Global.VIRTUAL\_HEIGHT). These walls should be tall enough to prevent dice from escaping.
3. A public method AddBody that takes a BodyDescription and returns the BodyHandle of the newly created physics body.
4. A public method RemoveBody that takes a BodyHandle and removes the body from the simulation.
5. A public Update method that takes a float deltaTime and steps the simulation forward in time.
6. Public properties to access the Simulation and BufferPool instances, as they will be needed by other classes.

You must output the complete, full code for the PhysicsWorld.cs file. Do not use comments like // ... or // rest of code is unchanged. Ensure all necessary using statements for BEPUphysics and MonoGame are included.

#### **Step 2: The Visual Representation of a Die**

**Prompt:**

Act as an expert C# and MonoGame developer specializing in 3D graphics and physics integration. Your next task is to create a class that handles the visual rendering of a single 3D die.

Create a new C# file named RenderableDie.cs. This class will not contain any physics logic; it is purely for rendering. It will hold a reference to a MonoGame Model and manage its position, orientation, and drawing.

The class must include:

1. A constructor that accepts a MonoGame Model object.
2. A public World property of type Matrix to store the die's position and rotation in 3D space.
3. A Draw method that accepts the camera's view matrix and projection matrix.
4. Inside the Draw method, you must iterate through each ModelMesh and ModelMeshPart of the Model. For each part, you must configure its BasicEffect with the correct World, View, and Projection matrices. Enable default lighting and texturing on the effect. Finally, call the Draw method on the mesh part's VertexBuffer and IndexBuffer.

You must output the complete, full code for the RenderableDie.cs file. Do not use comments like // ... or // rest of code is unchanged. Ensure all necessary using statements for MonoGame are included.

#### **Step 3: The Main Orchestrator**

**Prompt:**

Act as an expert C# and MonoGame developer specializing in 3D graphics and physics integration. Your task is to create the main orchestrator class, DiceRollingSystem, which will manage the entire dice rolling process from start to finish. This class will combine the physics from PhysicsWorld and the rendering from RenderableDie.

Create a new C# file named DiceRollingSystem.cs. This class will be the public-facing interface for the dice system. It will use the "Render to Texture" technique to keep the 3D rendering separate from the main 2D game.

The class must include:

1. Fields to store the GraphicsDevice, the PhysicsWorld, a list of active RenderableDie objects, and a dictionary mapping physics BodyHandle to its corresponding RenderableDie.
2. A RenderTarget2D for off-screen rendering of the 3D dice scene.
3. A 3D camera defined by View and Projection matrices. The camera should be positioned above the "table" looking straight down (orthographic projection is a good choice for this).
4. A Model field for the die model and a Texture2D for its texture.
5. An Initialize method that takes a GraphicsDevice and ContentManager. This method will:
   * Create the PhysicsWorld.
   * Create the RenderTarget2D matching the virtual screen dimensions.
   * Load the die.fbx model and die\_texture.png texture.
   * Set up the camera matrices.
6. A public Roll method that takes an int for the number of dice. This method will:
   * Clear any existing dice from the previous roll (both from the renderer and the physics world).
   * For the requested number of dice, create a new physics body (a box shape) and a new RenderableDie instance.
   * Give each new die a random starting position just off-screen and a random initial linear and angular velocity to make it tumble into view.
   * Add the new die to the physics world and the list of renderable dice.
7. A public Update method that takes a GameTime. This method will:
   * Step the PhysicsWorld simulation forward.
   * Loop through all active physics bodies. For each body, get its latest position and orientation from the simulation.
   * Update the World matrix of the corresponding RenderableDie to match the physics transform.
   * Include a public property IsRolling that is true if any die is still moving, and false otherwise. You can check this by examining the linear and angular velocities of the physics bodies.
8. A public Draw method that takes a SpriteBatch. This method will:
   * Set the GraphicsDevice to render to the RenderTarget2D.
   * Clear the render target to Color.Transparent.
   * Set the GraphicsDevice.DepthStencilState to DepthStencilState.Default to enable 3D depth.
   * Call the Draw method on every RenderableDie instance.
   * Reset the GraphicsDevice to render back to the main screen.
   * Use the passed-in SpriteBatch to draw the RenderTarget2D's texture over the entire screen.

You must output the complete, full code for the DiceRollingSystem.cs file. Do not use brevity comments. Ensure all necessary using statements are included.

#### **Step 4: Reading the Result**

**Prompt:**

Act as an expert C# and MonoGame developer specializing in 3D graphics and physics integration. Your task is to create the logic for determining the result of a dice roll and to integrate this logic into the DiceRollingSystem.

First, create a new static helper class in a file named DiceResultHelper.cs. This class will contain a single static method, GetUpFaceValue, which takes a die's final orientation Matrix as input. Inside this method, you will use the dot product to determine which face is pointing up. The world's "up" direction is Vector3.Up. You will compare this vector to the die's local axes, which can be extracted from the orientation matrix (matrix.Up, matrix.Down, matrix.Left, etc.). The local axis with the highest dot product (closest to 1.0) is the face pointing up. The method should return an integer from 1 to 6 based on a standard D6 layout. For example: matrix.Up (+Y) is 6, matrix.Down (-Y) is 1, matrix.Right (+X) is 3, matrix.Left (-X) is 4, matrix.Forward (-Z) is 2, and matrix.Backward (+Z) is 5.

Second, modify the DiceRollingSystem.cs file you created previously.

1. Add a public event public event Action<List<int>> OnRollCompleted;.
2. In the Update method, when you detect that all dice have stopped rolling (i.e., IsRolling transitions from true to false), you must:
   * Create a list of integers for the results.
   * Loop through each die's physics body.
   * Get its final orientation matrix.
   * Call DiceResultHelper.GetUpFaceValue with this matrix to get the result.
   * Add the result to your list.
   * Finally, invoke the OnRollCompleted event with the list of results. Make sure to only invoke this event once per roll.

You must output the complete, full code for both DiceResultHelper.cs and the modified DiceRollingSystem.cs. Do not use brevity comments.

#### **Step 5: Integration into the Game**

**Prompt:**

Act as an expert C# and MonoGame developer. Your task is to integrate the completed DiceRollingSystem into the existing game structure. You will need to modify two files: Core.cs (the main game class) and TerminalMapScene.cs (the primary game scene).

First, modify Core.cs:

1. In the Initialize or LoadContent method, create an instance of DiceRollingSystem.
2. Register this instance with the ServiceLocator so it can be accessed globally.
3. Call the diceRollingSystem.Initialize() method, passing it the GraphicsDevice and Content.

Second, modify TerminalMapScene.cs:

1. Add a field to hold a reference to the DiceRollingSystem.
2. In the constructor or Enter method, retrieve the DiceRollingSystem instance from the ServiceLocator.
3. Subscribe to the diceRollingSystem.OnRollCompleted event with a new handler method. This handler method should take the List<int> of results and publish a GameEvents.TerminalMessagePublished event to display the roll result in the game's terminal (e.g., "You rolled: 4, 2 (Total: 6)").
4. In the Update method, call diceRollingSystem.Update(gameTime).
5. In the Draw method, after all other drawing is complete (but before any final UI overlays if necessary), call diceRollingSystem.Draw(spriteBatch).
6. For testing purposes, add temporary input handling to the Update method. If the 'R' key is pressed, call diceRollingSystem.Roll(2) to roll two dice.

You must output the complete, full code for both the modified Core.cs and TerminalMapScene.cs files. Do not use brevity comments.