1 Introduction

This project is based on the 'physics-based game' template, which requires a short, but compelling game using physics modelling as a key element of gameplay.

My proposed design will be based on simulating a tabletop board game. Board gaming is an area that holds especial interest for me, as a structure around which to base social engagements, and for the mental – and occasionally physical – challenges such games provide.

While board games are inherently physical items, a subset – known as dexterity games – make explicit use of their physicality. Such games may challenge players to manoeuvre components using manual dexterity with varying objectives, such as building towers, accurately flicking pieces around a game space, or carefully manoeuvring components using tools. Well-known examples of games in this genre include *Jenga*, *Operation*, and *Subbuteo*. This focus on the physical interaction of game components may make a dexterity board game a suitable influence for a project conforming to the provided template of a 'physics-based computer game'.

The project template specifies that the end product be easy to pick up and play. It is therefore assumed that the users (players) will not necessarily be 'hardcore' gamers, familiar with the conventions and control schemes of modern PC gaming. The users may, or may not, have familiarity with the type of physical game upon which the software will be based. Both the gameplay, and the methods of control and interaction must therefore be reasonably intuitive, or quickly learnable. The users may not have access to advanced computer hardware, and may not have use of control pads or other gaming-specific input methods. It will be important that the software can run on a reasonably specified machine, and can be used with mouse-and-keyboard inputs.

2 Feature prototype

One of the key requirements for this project is the ability for the player to pick up and control a game piece with six degrees of freedom. The first prototype thus focuses on implementing a control scheme to achieve this.

The prototype was created within Unity; a simple scene consisting of a camera and light source, with a floor plane and several game pieces was created. The game pieces will eventually be constructed as more complex rendering and collision meshes, but for this prototype primitive shapes were used.

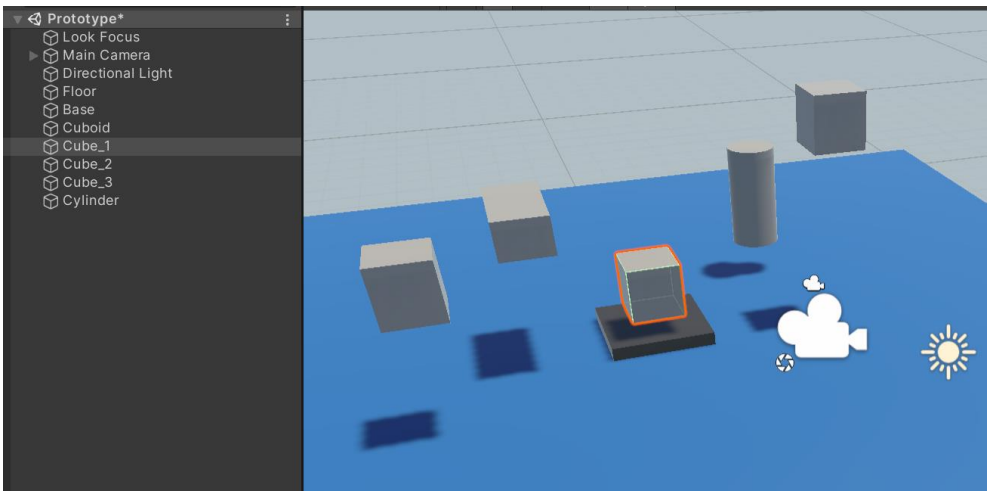

Figure 8: prototype scene

The player does not require an avatar within the game world, so the camera is controlled directly using the WASD-and-mouse scheme familiar to players of modern games. This is implemented in a C# script attached to the camera. The script is adapted from an example included one of the template projects bundled with Unity 2019.

A second script 'GrabObject' is also attached to the camera. This script allows the player to 'pick up' a game object, and to control it while it is in the 'held' state. When the player clicks the primary mouse button, a ray is cast through the cursor location. If an object tagged as a 'game piece' is hit, that piece is picked up. The object is childed to an empty transform (*holdFocus*) which is itself a child of the camera, and it is set to ignore gravity. While in this

state a game object will be held at a point in front of the camera, and thus be moved around the game world as the player moves the camera.

```
private void grabObject(GameObject target)
{
    //check target has an RB
    if (target.GetComponent<Rigidbody>())
    {
        //assign grabbed object
        heldObject = target;
        heldObjectRB = target.GetComponent<Rigidbody>();

        //set physics parameters
        heldObjectRB.useGravity = false;
        heldObjectRB.drag = 10;
        heldObjectRB.constraints = RigidbodyConstraints.FreezeRotation;

        //parent held object to grab area
        heldObjectRB.transform.parent = holdFocus;

        //set the object's tag
        heldObject.tag = heldTag;
    }
}
```

Figure 9: grabObject function within the GrabObject class

Attaching an object to the camera in this way allows it to be translated freely on the X and Z axes. There is a limited movement on the Y axis by rotating the camera, but to allow full movement the mouse scroll wheel is used. When scrolling while holding a game piece, the location of the '*holdFocus'* transform is moved with respect to the camera, moving the held object towards or away from the player.

Game piece rotation is achieved by activating a 'rotate' key, set as the space bar. While the space bar is held, the game switches into 'rotate' mode, in which the normal camera controls are disabled. This is achieved using the event system: the GrabObject class includes *onRotateStart* and *onRotateStop* events, to which the CameraController class subscribes and adds functions to disable and reenable camera controls. These events are invoked within the GrabObject class while the rotate command key is held. In this state, the GrabObject class instead interprets mouse XY movement and scrolling to rotate the held object around each of its axes.

```
private void rotateObject()
{
    //get mouse movement
    Vector3 mouseMovement = new Vector3(
        Input.GetAxis("Mouse Y") * -1,
        Input.GetAxis("Mouse X") * -1,
        Input.mouseScrollDelta.y * 5
        );

    heldObjectRB.transform.eulerAngles += mouseMovement;
}
```

Figure 10: rotateObject function within the GrabObject class

These two classes, along with the Rigidbodies and Colliders of the game objects, allow for full control of the game pieces with six degrees of freedom, which was the primary goal of this prototype.

Additional functionality important to the final product was also trialled. It will be important to know, for each game piece, whether it is successfully added to the player's growing structure, or if it falls off and is no longer counted. Each game piece starts with the 'unstacked' tag. A script is attached to each piece to detect collisions with other game objects. On a collision, the tag of the other object is read: if it is the structure base, or any piece tagged as 'stacked', the active piece is also tagged as 'stacked'. If a piece collides with the floor, it is tagged as 'grounded'.

At the same time, the game piece material is updated to give a visual indication of its status: grounded, held by the player, or stacked in the structure.

```
private void OnCollisionEnter(Collision collision)
{
    //do nothing while piece still held by player
    if (transform.gameObject.tag == heldTag)
    {
        return;
    }

    string otherTag = collision.gameObject.tag;

    //flag as grounded
    if (otherTag == floorTag)
    {
        transform.gameObject.tag = groundedTag;
        thisRenderer.material = unstackedMat;
        onStackChanged?.Invoke();
        return;
    }

    //flag as stacked if not grounded, and connected to base or existing stack
    if (
        (otherTag == baseTag || otherTag == stackedTag)
        && transform.gameObject.tag != groundedTag
        )
    {
        transform.gameObject.tag = stackedTag;
        thisRenderer.material = stackedMat;

        //broadcast event
        onStackChanged?.Invoke();
    }
}
```

Figure 11: updating a game piece tag and material on collision, within the PieceStacked class

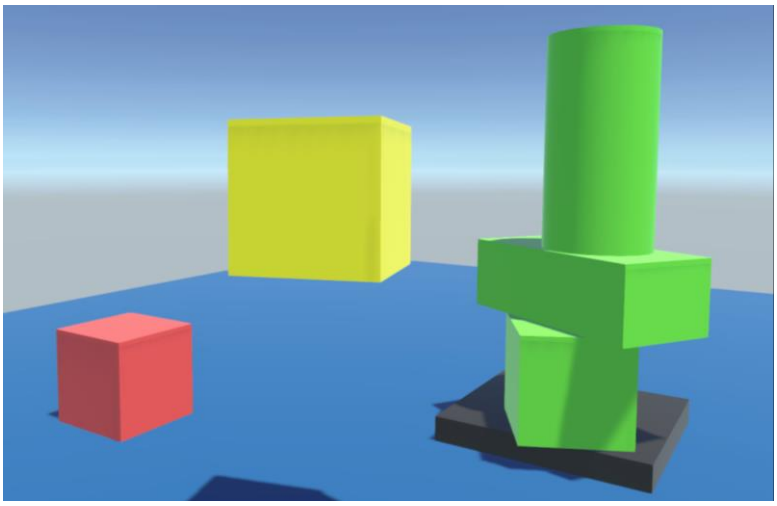

Figure 12: Game pieces in various states indicated by their material colour: grounded (red), held by player (yellow), stacked (green)

The final piece of functionality included in the prototype is the calculation of the total height of the player's structure. This will be necessary to determine which player is leading, and has won, a round.

A GameController script is attached to the camera. This class subscribes to the OnStackChanged event defined in the PieceStacked script attached to each game piece, and adds a CalcMaxHeight function to the event. Whenever the a piece is added to the player's structure (and tagged as stacked), or a piece touches the floor (having potentially fallen from the structure), the function loops through all stacked game pieces and finds the maximum Y value of the highest axis-aligned bounding box (AABB). The nature of the AABB means this value will always be the highest point on of the piece's collider within the world space. The results are output to the debug console, but will be exposed to the players through the game interface.

```
private void calcMaxHeight()
{
    float maxHeight = 0f;

    GameObject[] stackedList = GameObject.FindGameObjectsWithTag(stackedTag);

    //loop through each object in the stack
    foreach(GameObject piece in stackedList)
    {
        //update max height with top of bounding box
        maxHeight = Mathf.Max(maxHeight, piece.GetComponent<Collider>().bounds.max.y);
    }

    Debug.Log("Stack height: " + maxHeight.ToString("0.0"));
}
```

Figure 13: calcMaxHeight function within the GameController class

The features implemented in the prototype serve to prove the feasibility of the first part of the project plan: a single-player implementation of the basic game mechanics. This will require building upon to flesh out the product into a viable game: custom game pieces with collision meshes, hot-seat multiplayer functionality, player displays, and a menu system.

While the prototype does implement the features planned, it is not without issues. The camera moves very jerkily under the player controls, which makes precisely placing a held game piece difficult. Object rotation is not particularly intuitive; additional experimentation with this system is required. Further

research into how this problem has been tackled by other games should prove helpful with this. The interaction of the camera and game piece classes occasionally exposes intended behaviour, such as continued disabling of camera controls after a piece is dropped. Cases such as this will hopefully be caught by the playtesting regime planned throughout the development phase.

There has been limited opportunity for third-party testing of this prototype, but one other person has experimented with the software in its current state. This person is not a regular gamer, and has little familiarity with the WASD-and-mouse control scheme. Without instruction they were unable to effectively interact with the game world. This emphasises the need for, at the very least a menu listing the game controls. Ideally an interactive tutorial would be included in the final product, but this is likely to be beyond the scope of the resources available.