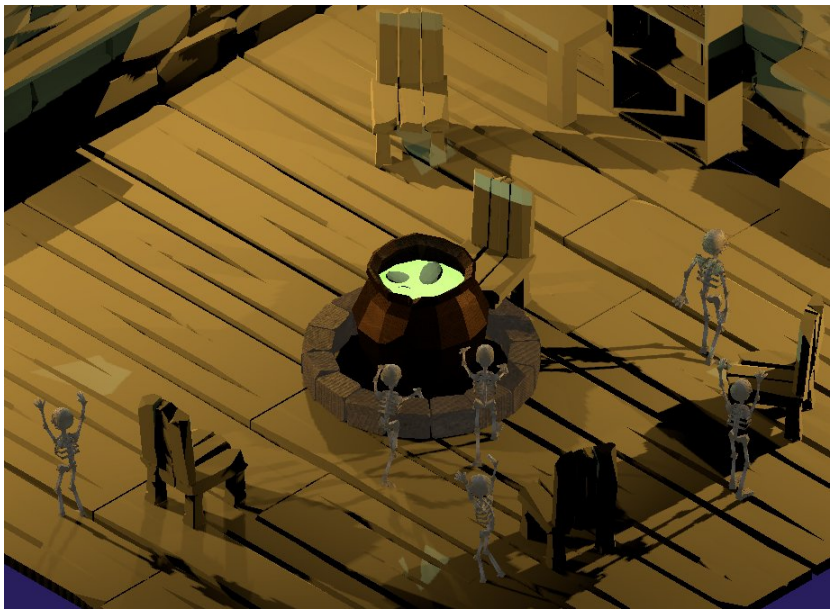
Interactive Graphics
Final Project
The Chair Game

Emanuele Giacomini 1743995
emanuele.giacomo@gmail.com
giacomini.1743995@studenti.uniroma1.it

July, 2020



1 Introduction

In 'The Chair Game' the user's objective is to survive by begin able to "seat" on a chair before all the other spooky foes occupy them. In order to occupy a place, the user needs to touch a seat during the rush phase of the game.

In the following document I will explain the development of my game in all its aspects. The discussion will articulate on the following arguments:

- Game Definition
- Skeleton model
- Environment
- Models and instances
- Collisions
- Planning and animations
- User Interaction (UI)

The project heavily relies on the **Three JS** library and adopts *noisejs* for perlin noise generation and finally *dat.gui* for the graphical user interface (GUI).

2 Game definition

The game develops in multiple turns and regards a set of n skeletons (of which one is controlled by the player) whose task is to sit on $n-1$ available chairs. Once all the available seats are occupied, the last standing unit gets eliminated, and a new turn starts, this time with $n-1$ skeletons and $n-2$ chairs. This process keeps repeating until only one skeleton survives, hence winning the game. The user is able to configure the number of enemies present in the play, and their respective difficulties between:

- easy
- hard
- unfair

of which differences regards only the physical characteristics of the enemies, and not their character traits.
The whole game phases can be represented as a finite state machine (FSM) presented in the next figure. In practical terms, the state is represented as an unsigned integer ranging from 0 to 2 and each transition is triggered by game-related events, namely:

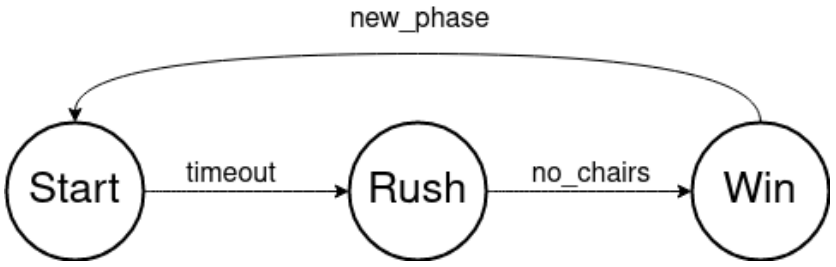


Figure 1: Game FSM

- **timeout** happens when the $phase_0$ time limit is reached, thus enabling the **rush** state.
- **no_chairs** happens when the last available chair is captured by one skeleton, ending the current turn.
- **new_phase** is triggered when the game has ended, and if **at least one chair is still available**

Each state's transition will trigger different reactions from the dynamic elements of the scene, thus "moving" the game forward.

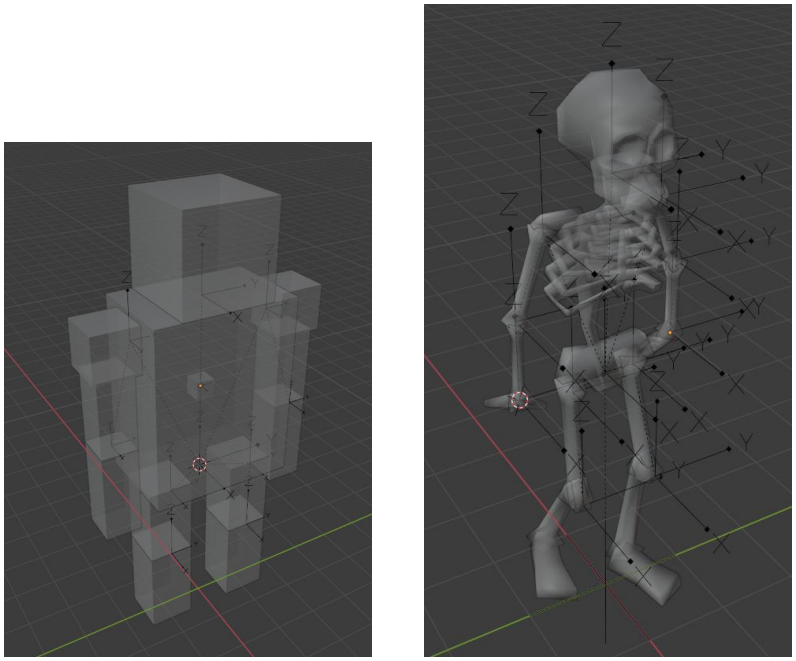
3 Skeleton model

Before using the final production skeleton, an easier model was built in *Blender* in order to test AI behaviour and animations. As requested, the model (named *simguy* or simulation-guy) is composed by 7 layers, namely:

- *baseframe*
- *torso*
- *head*
- *shoulders*
- *lower arms*
- *upper legs*
- *lower legs*

The hierarchy is defined by assigning parent relationship between the different parts (eg. upper legs are parents of lower legs and torso is parent of upper legs). The main body part however, is represented by the *baseframe*: an hidden element whose position and orientation characterise the unit itself. Initially I planned to use the armature framework in order to generate a skeleton for the

model, however this approach was discarded due to the additional challenges of creating animations on the *Three JS* framework.
Each body part's frame was carefully put in place in order to create natural-like joints. Finally the skeleton model was also provided with hands (which were not used for animation for time related reasons).



(a) Simguy (b) Skeleton

Figure 2: The two dynamic objects used in the game. Notice that topologically the models are quite similar.

4 Models and Instances

In order to import these models in the game, I've used the *GLTFLoader* (provided by *THREE JS*) Loader which was able to load *gltf* (GL Transmission Format) exported models into the *THREE* environment. Each object is then mapped into a *THREE.Group* or *THREE.Object3D* depending on its scope. Since both chairs and skeletons should be loaded multiple times, I've decided to setup an instancing system which would first load all the models, and then clone the ones who should be instances more than once. The system relies on two script-based dictionaries, namely:

- *models* contains for each model's name, the path to reach its model
- *units* contains for each unit, the path to its model, its main body part (mesh), the on-load-complete function, position and orientation and finally the optional planner.

```
const models = {
    'map' : {file: 'map/map.gltf'},
    'chair':{file: 'chair/chair.gltf'},
    'skeleton':{file:'skeleton/scene.gltf'}
}

var units = {};
```

In the main script, I've defined 4 functions with the scope of adding each possible entity in the game:

- addMap() : Load a *map* instance
- addAI() : Load a *skeleton* instance with *ai* planner
- addPlayer(): Load a *skeleton* instance with *human* planner
- addChair(): Load a *chair* instance

Once the units dictionary is filled, the *loadModels* function may be executed, loading models and instancing all the units. As described before, each unit may have an on-load-complete function that will finalise its form (eg: initialize map lights for the map instance, or planner/animation-handlers for skeletons). Once all the units are loaded, the *render* function should be called in order to visualise the whole scene.
Now that the scene is set up, I'd like to talk about collisions, which I've decided to handle personally, without the aid of *PhysiJS*, an extension of *THREE JS* for physics handling.

5 Collisions

In the Chair Game, collisions are fully handled by an own-made collision system based on the *THREE JS* library. It is important to mention that the system can be used on each *Object3D* element present in the scene.
The collision system works by continuously checking for intersections between *Box3*s ideally representing the objects bounding boxes. In order to resolve collisions as quickly as possible, the bounding boxes related with moving objects were stored in a separate array named *dbboxes* (or dynamic bounding boxes) and are the only ones who are continuously tested for intersection (with the assumption that other elements in the scene are all static).

During the rendering function, all collisions are found by testing if any dynamic bbox intersects any static bbox (collisions between dynamic bboxes was not included) and eventually storing information relative to the collision into the queue *collisions*. Once all the possible collisions are tested, the system begins resolving each collision separately. For each collision, the system locally computes the inverse direction *iNormL* between the moving object's centre and the collided object's centre. In order to nullify the collision, the object should be moved along *iNormL* of a certain distance which produces different effects: a small distance may not fully solve the collision, hence causing a larger number of events to be handled while a large distance may produce an unnatural effect of

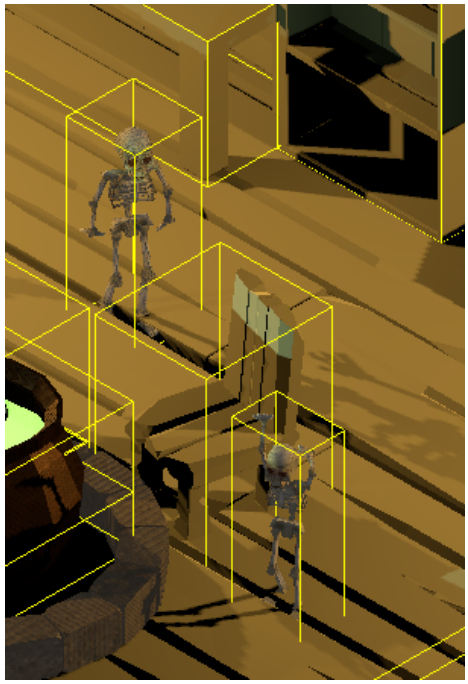


Figure 3: Bounding boxes for different elements of the scene

repulsion from objects which may become annoying from the user's perspective. After multiple tests, I was able to obtain a quite enjoyable distance proportional to the magnitude of the collision space, obtained by extracting the 3 spatial dimensions (*Vector3*) of the intersection box and then projecting it towards the *iNormL* vector. This metric also takes into account the direction of intersection, thus producing a more natural collision behaviour.

```
function evalCollision(object, obbox, ibbox) {
    // obbox : Object BoundingBox
    // ibbox : Intersection BoundingBox
    const iSize = new THREE.Vector3();
    const iCenter = new THREE.Vector3();
    const oCenter = new THREE.Vector3();
    // Extract data from obbox and ibbox
    ibbox.getSize(iSize);
    ibbox.getCenter(iCenter);
    iSize.setY(0);
    obbox.getCenter(oCenter);
    // Compute global inverse collision direction
    const eDir = new
    ↪  THREE.Vector3().copy(oCenter).sub(iCenter).setY(0).normalize();
    const eDist = new
    ↪  THREE.Vector3().copy(iSize).projectOnVector(eDir);
    // Apply world to local transformation to eDir
    const rMat = new
    ↪  THREE.Matrix4().extractRotation(object.matrix);
    rMat.getInverse(rMat);
    // Push the object in the local eDir direction by eDist
    object.translateOnAxis(eDir.applyMatrix4(rMat),
    ↪  eDist.length());
}
```

An interesting notice regards the fact that y-component for both intersection size vector and escape direction vector are set to 0 in order to avoid any translation component on the y- axis component which defines the plane on which the game is executed. Once collisions are handled, its possible to define both the animation handlers and the planners, explained in the following sections.

6 Animation Handler

Conceptually each *Object3D* should be able to perform animations during the execution of the game without begin asked directly to output the next pose at each rendering update. The designed animation handling system is an independent system of the game which keeps track of all the animations currently in execution and decides the next pose based on both external and internal parameters. In the current game phase, three *AnimationHandler* classes exists:

- AnimationHandler : Parent class containing standard methods used to handle animations
- TorchAnimation : Handler for the light and flame elements present in the scene
- SkeletonAnimation : Handler for skeleton hierarchical models

The *AnimationHandler* class exposes a single generalised method *AnimationHandler.update* used by the rendering routine to query the handler, hence updating che current animation state. In particular the *AnimationHandler.update* function implements the framework required to animate the specific object. For simplicity I've decided not to implement animations through the keyframe interpolation method proposed during the lectures. Instead by linearly combining a set of periodical functions (sine, cosine, etc) it was possible to design most if not every animation currently developed.
More explicitly, the most complex handler for the game, the *SkeletonAnimation* handler, is able to replicate 7 different animations for walking, running, dancing, sitting and so on. The programmer may change the current pose of his skeleton by calling the method *SkeletonAnimation.setPose* function which takes as argument the name of the target animation as a string.

```
function onGuyLoaded(object) {
    // generate bounding box
    // ...
    // Add random seed [0, 1000] for animation offset
    // Install animation handler
    const mixer = new ANIM.SkeletonAnimation(object, 'idle',
    ↪  Math.floor(Math.random() * 1000));
    object.unit.animationMixer = mixer;
    animations.push(mixer);
    // ...
}
```

A parallel work done with animation handling was **Path Planning** for dynamic objects such as skeletons, which will be further explained in the next section.

7 Planners

Each mobile unit should autonomously or through player attitude, be able to reason and move accordingly to the game's objective. The *Planner* class and its children are defined in order to plan and control each unit's action throughout the game execution. The two relevant planners explained here will be the ones related to the player and the skeletons, namely:

- HumanPlanner
- ChairFinder

The **HumanPlanner** is used as a bridge between the UI (User Interface) and the unit's motion: it works by initially creating two event listeners in the webpage for both catching *keyup* and *keydown* events on the WASD keys. Each combination of event plus key produces a positive/negative translational/rotational desired velocity on the unit, which is then applied at every updating step (rendering step).
The **ChairFinder** planner, is slightly more complex as it needs to simulate the behaviour of another player. In the construction phase, its possible to setup the planner's parameters, which also includes:

- speed
- angular speed

which are both used to define the difficulty of the AI itself.
In the first phase of the game, when chairs are not available yet, this planner will execute a **Random walk** by computing a random perturbation of its heading through **Perlin Noise** implemented in the *NoiseJS* library.

```
ChairFinder.update(time) {
    // ...
    if (this.state == 0) { // Initial Phase
        // Execute Random Walk
        const rnStep = noise.perlin3(this.object.position.x,
        ↪  this.object.position.z, time);
        this.object.rotateY(rnStep * this.aspeed * 5);
        this.object.translateOnAxis(new THREE.Vector3(1,0,0),
        ↪  this.speed);
    }
}
```

In case of collisions in this phase, the *ChairFinder* will also reflect its current heading on the collision normal, in order to favourite a circular trajectory in the scene:

```
ChairFinder.onCollision(collision) {
    // ...
    if (this.state == 0) {
        const object = collision.object;
        const iNormL = COLL.computeLocalCollisionNormal(object,
        ↪  collision.bbox, collision.ibbox);
        const newDir = new THREE.Vector3(1,0,0).reflect(iNormL);
        // rotate object towards newDir
    }
    // ...
}
```

Finally when entering in the second phase of the game, the planner will aim at its closest chair which is still not occupied by any other skeletons and upon reaching it, it will query the chair in order to occupy it.

```
ChairFinder.update(time) {
    //...
    if (this.state == 1) {
        // Find closest chair
        let tChair; // = get closest available chair
        if (tChair) {
            const oDir = getChairDir(this.object,
            ↪  tChair).normalize();
            const oDirLocal = worldToLocalRotation(oDir,
            ↪  this.object.matrix);
            // Compute angle between (1,0,0) and oDirLocal
            let astep = ...;
            // ...
            this.object.rotateY(-astep);
            this.object.translateOnAxis(new THREE.Vector3(1,0,0),
            ↪  Math.max(0.0, new
            ↪  THREE.Vector3(1,0,0).dot(oDirLocal)) *
            ↪  this.speed);
            // ...
        }
    }
}

ChairFinder.onChairCollision(collision) {
    if (this.winner) return; // if already won, do nothing
    const chair = collision.tobject.unit;
    if (chair.onQuery) {
        // If chair is available, occupy it and win
        if (chair.onQuery(chair)) this.winner = true;
    }
}
```

Notice that the chair collision handling routine is exactly mirrored in the *HumanPlanner* planner.