Custom Project Document

Herbert Henkel – 102568830

Code Link - https://bitbucket.org/102568830/cos30002-102568830/src/master/21%20-%20DOC%20-%20Custom%20Project%20D%20Level/TopDownGame/

Video Demonstration - https://www.youtube.com/watch?v=fS3dax6Ur5s&feature=youtu.be

Project description

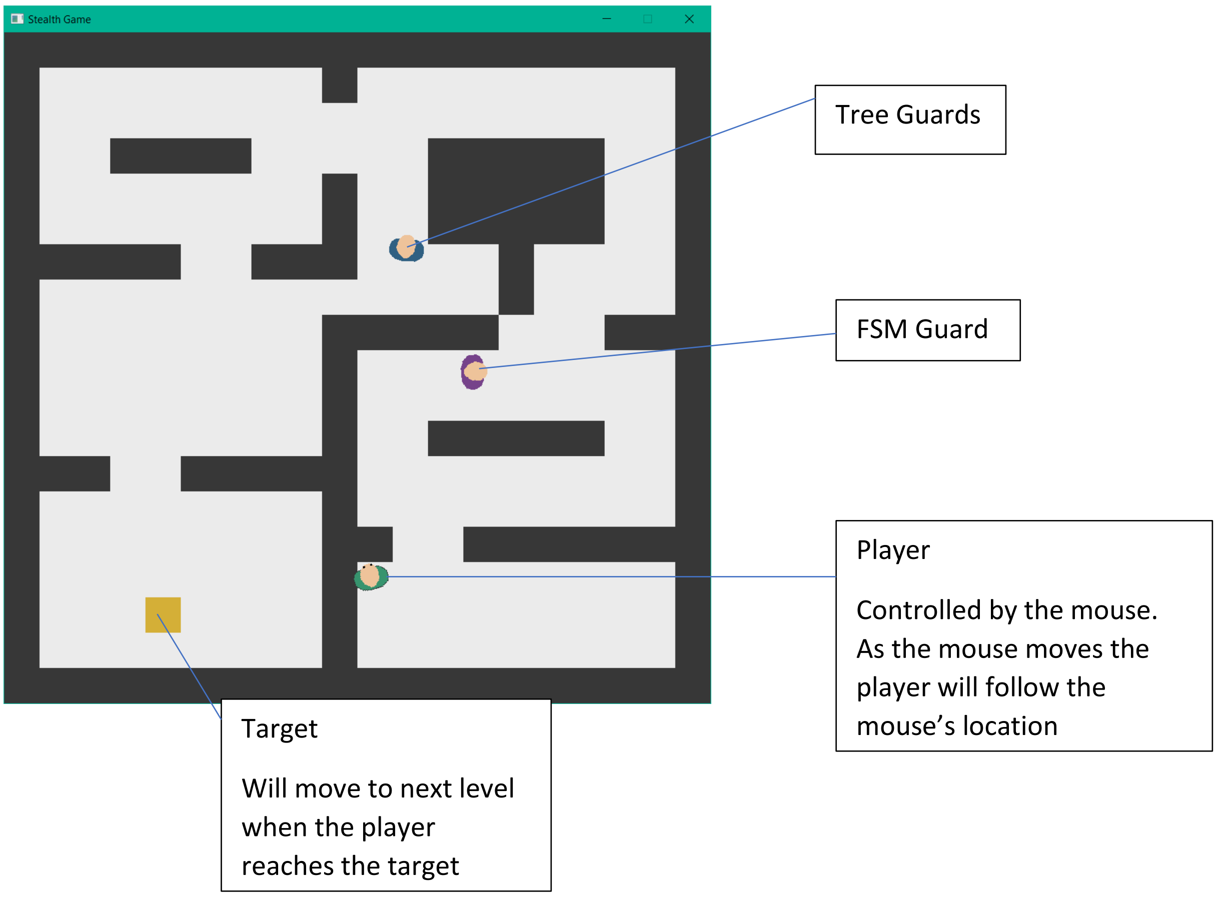
My custom project is a 2d top down stealth game where the player navigates a grid-based map sneaking around to reach a target, whilst sneaking around patrolling guards. The purpose of the project is to compare the techniques of behavior trees and finite state machines as structures for controlling an agent's behavior.

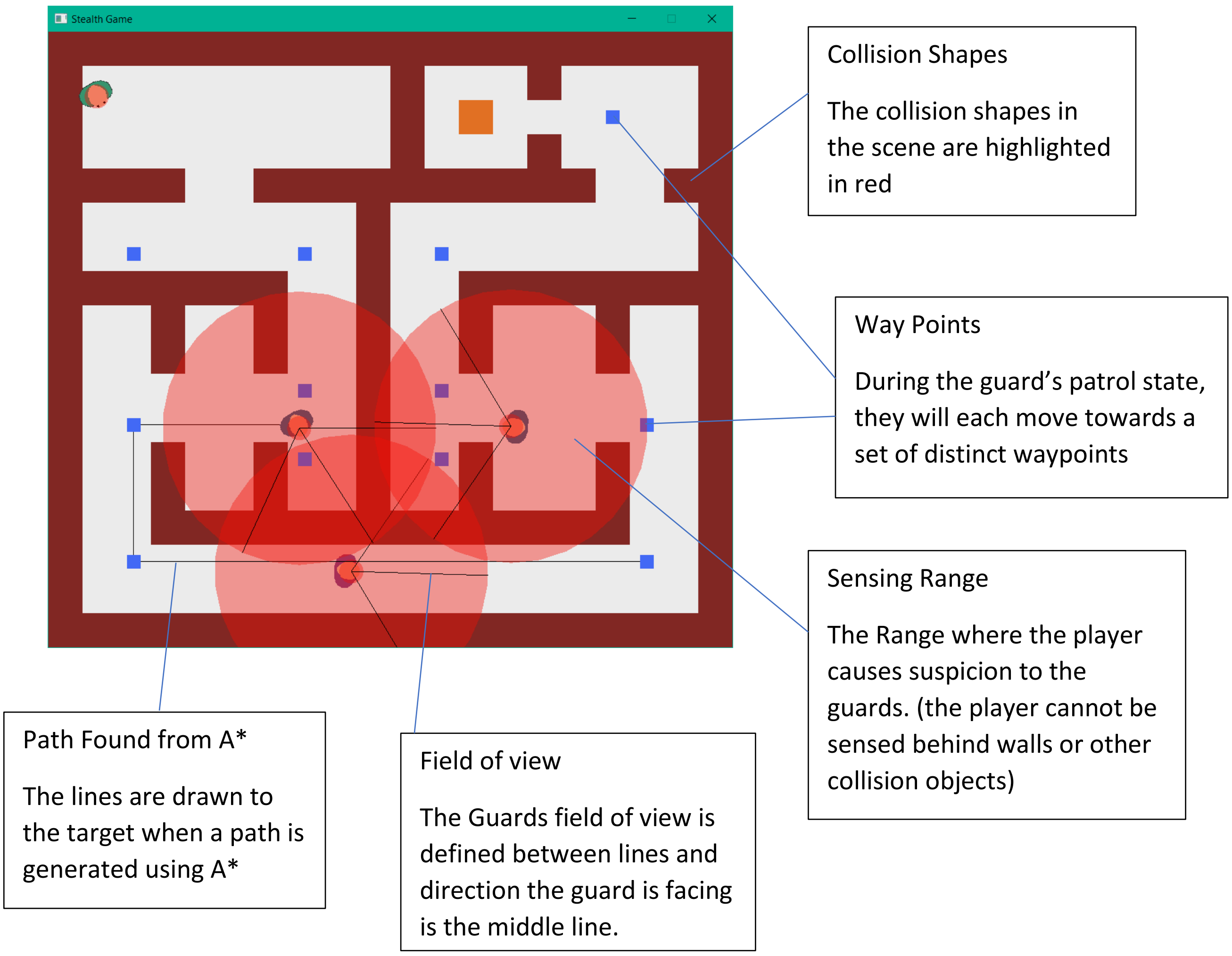
In order to demonstrate the two techniques, I created two guards that each implement one of the techniques as the primary method of controlling their behavior. Each guard is a different color to differentiate the two in the final product.

The other techniques on display in my implementation are steering force and A* path finding. In the project steering forces are used to move both the player and the agents around, I also used steering forces so to enable the agents to avoid each other when they were heading the same direction. I use the A* algorithm for path finding, to generate a path to each target during the waypoint of the agent's path.

Custom Project

Below I included two screenshots of my custom project's implementation the Normal mode and the Debug mode (accessed through pressing the space bar).


Stealth Game
Tree Guards
FSM Guard
Player
Controlled by the mouse. As the mouse moves the player will follow the mouse's location
Target
Will move to next level when the player reaches the target



Collision Shapes

The collision shapes in the scene are highlighted in red

Way Points

During the guard's patrol state, they will each move towards a set of distinct waypoints

Sensing Range

The Range where the player causes suspicion to the guards. (the player cannot be sensed behind walls or other collision objects)

Path Found from A*

The lines are drawn to the target when a path is generated using A*

Field of view

The Guards field of view is defined between lines and direction the guard is facing is the middle line.

Implementation

Project structure

My game was created using c# along with two library SFML for the window, graphics, and controls. As well as box 2d for the physics.

The structure of the project is there is a base class called Game. This class is a singleton, this design pattern is used to ensure that there is only one instance of this class is created, it is also used to allow the contexts of the game to be accessed across the project, like what the current scene is or give references to other game objects. The game holds, loads, and runs the current scene in the game.

In the next major component in the project are the scenes, the scenes are the levels, and hold all the game objects in the current scene. The scene also iterates through and runs each game object.

The next major component and the most widely used within the project is the abstract class GameObject. GameObjects are everything that exists within the scene from the tilemap to the player and enemies.

Collision

This section of the report I'm going to talk about how I handled the collision in my game, whilst not directly related to game AI it was a portion of the development that I did put a considerable amount of time into and learnt a lot from, so I think it is worth mentioning.

For collision I created an abstract class CollisionShape all the different collision shapes inherit from this, and the interface ICollidable, all the colliding game objects inherit from this interface. All the colliding game objects need to implement the collision shape component and use that shape to define the collision region of the game object, as well as to detect collision.

For Collision Detection each shape implements the collision detection for that specific shape (The code for the collision detection is based off Jeffery Thompsons website http://www.jeffreythompson.org/collision-detection/table_of_contents.php).

To handle the players collision, I initially tried writing the code the handle the collision myself and part of my need for the collision was for colliding objects to slide against each other. My worked well when the collision just rectangles but when I tried adding more collision shapes like circles, the smooth sliding did not carry over. After a lot of fine tuning and tweaking of the collision handling. I eventually realized that I would need to use a library to handle the collision, so I used the library Box2D to handle the collision. I hadn't used this library before and was unsure how will it fit with my current implementation, however it fit well into the code I had already written and my collision shapes worked well as wrappers for the Box2D collision shapes.

Guard line of sight and Shooting

To handle shooting and whether the player is in sight I use very similar techniques involving the line collision shape.

To detect if the player is in sight or sensing range of the guard I check if the player is within a the radius of the sensing range, however using this technique on its own has a big issue, the player can still be sensed even if they are hiding behind a wall. For a stealth game where hiding behind walls from the guards is a big part of it. To Address this problem, I added a new method to the Collision line object that return a list of collision objects that the line collides with. So, when the player is withing the sensing range of the guards, the guard will create a line between its position and the players position and if there is another object on this line the guard cannot see the player.

For Shooting I used a similar technique by drawing a line that represents the guards bullet trajectory and if it collides with the payer I check that the player is the closest collision object to the guard before the player takes damage.

Path finding

To handle path finding I created the class Path that handles the pathfinding. The path class consists of a list of points that represent the path. So, navigating the path consists of moving to a point along the path and upon arrival incrementing to the next point.

Each guard implements 2 paths as components, the first component in the guard is the Path the guard follows during its patrol, the points for this path are defined in the map files. The second path component is the path the target this is the path that utilizes A* for navigation, this is the path to the current target whether the target is a way point in the guards patrol or the players location.

For better efficiency during the pathfinding I check whether the path is clear by creating a line collision object between the guards position and the targets position and checking for collision between the line and the tile map and only call the A* algorithm if a wall between the target.

Steering forces

All the movement in the game utilizes steering forces.

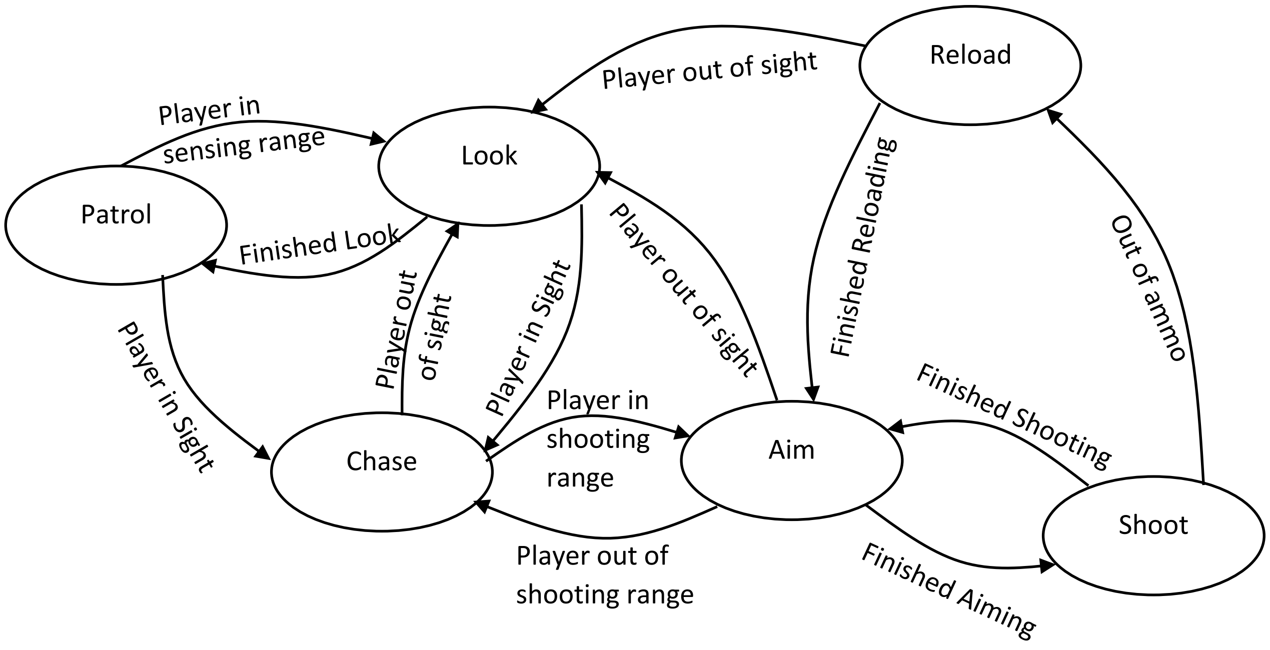
The players movement is the arrive steering force towards the location of the mouse.

The Guards movement is seek steering force towards its target whether it's the player or its target along the path.

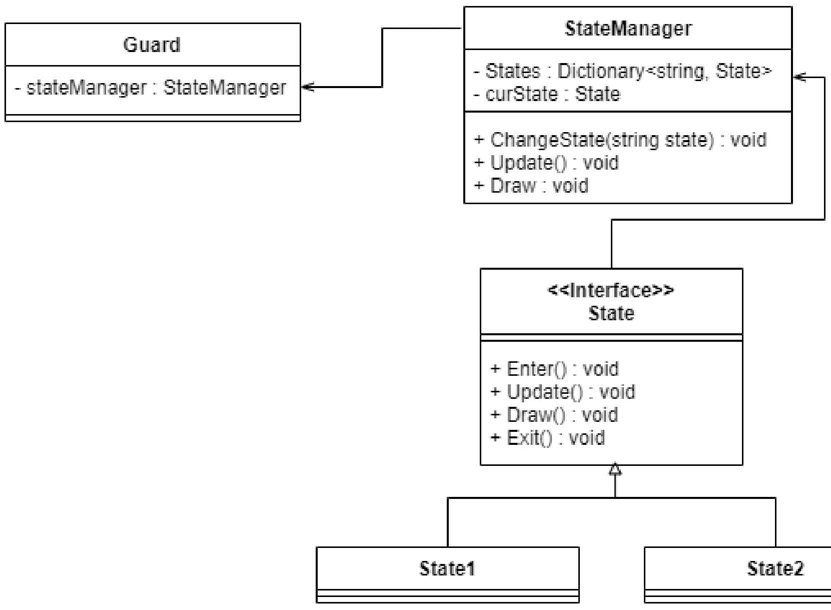
Another thing I wanted the guard to do during its patrol state is to do collision avoidance between other guards so they don't just wander into each other. To achieve this, I played around with and tweaked a combination of steering forces. The first force I looked at was the separation steering force the result of this is when the guards got close to each other they will repel like magnets but this looked unnatural as the paths around each other was quite large, the second steering force I looked at was collision avoidance (a steering force I looked at during a spike extension based off Fernando Bevilacqua article https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777), after adding this steering force the avoidance was a lot smoother and more subtle, however a lot of the time during collision avoidance the guards were trying to avoid each other in the same direction, to address this I added the force alignment so during the collision avoidance the guards align in the opposite directions so they don't collide whilst trying to avoid each other.

Finite State Machine

One of the Guards in my game implements a finite state machine to handle the behavior. The State diagram the guard is below.

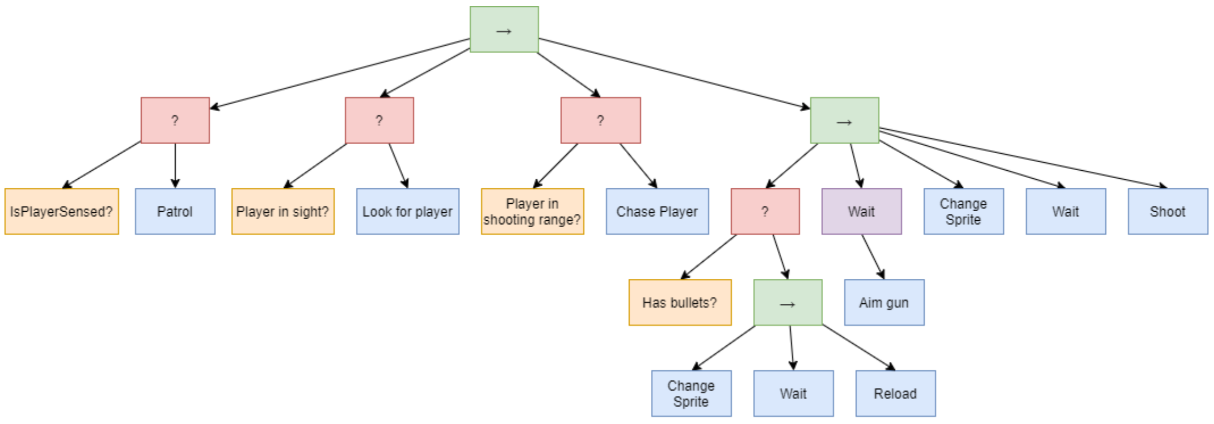


To implement the finite state machine, I used the following design pattern.



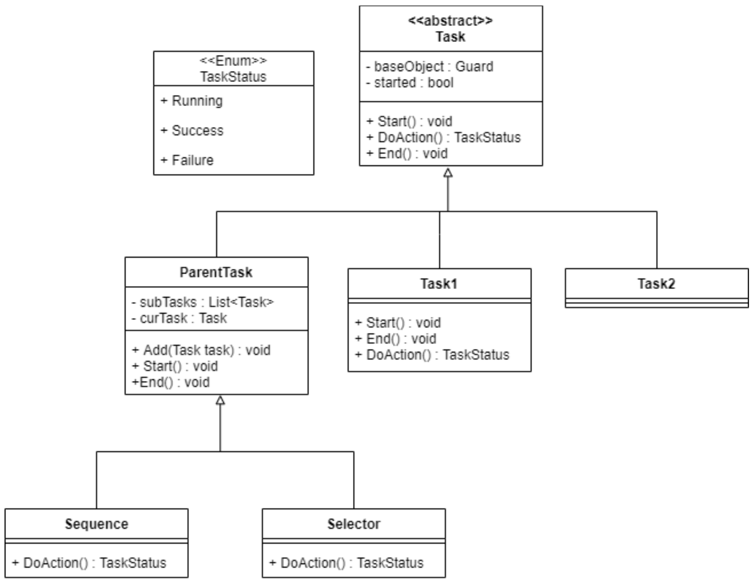
Behavior Tree

Before this subject I have never implemented a behavior tree before, so I had to do research to learn about how they worked and how they were used. From what I learnt I created a plan to convert the behavior from the FSMGuard to a behavior tree. The plan is shown below.



In behavior trees there are 3 kinds of nodes leaf nodes, parent node, and decorator nodes. Leaf nodes are used the tasks that are at the and of each branch, there are type kinds of leaf nodes, Actions and Conditionals. Actions are when the node performs an action, example from my tree above is the patrol task or the look for player task. Conditionals are when the task tests for a condition and will never return running. Parent nodes are a node that contain multiple child nodes, in my implementation there are two kinds of parent nodes Selectors and Sequence. The final node type is decorators, decorators are tasks with a single child, and the decorator modifies the result of the underlying task, in my implementation I have one decorator node Wait. The wait node will return the value of the underlying node until a certain amount of time has passed then it will return successful.

The design pattern I used to create the behavior tree is shown below



When implementing the behavior tree, a big issue I had was handling interruptions of other tasks and knowing to end a task it was interrupted. To do this I created a queue of active tasks and if the queue has more than one task it means the previous task was interrupted so I need to end the previous task.

Comparison of behavior trees and Finite state machines

After implementing both techniques behavior trees and finite state machines I now feel like I have a good understanding of the differences between the two and their pros and cons.

What I found to be the biggest difference between the two techniques is where the decision making is made, with behavior trees the decisions and test for changing behavior is built into the structure itself, made up by more modular components. And with finite state machines the decision making is done within the action of each state by a series of checks. The result of the structure of the tree being how decision making is done allows for a larger amount of flexibility and ease to change the behavior by small tweaks to the design, whereas with state machines the decision to change state being within the state create large dependencies between each state and to make changes between the states can cause large amounts of change within the code to be made.

Another difference between the two techniques is the modularity and reusability of the components used for the techniques. With state machines each state is for one specific purpose and only can be used once, for example two different enemies will not be able to share the same state because of dependencies between the states. But with behavior trees the components have a lot more modularity and reusability between the tasks. For example, in my behavior tree, I implement the tasks change sprite or wait multiple times, these tasks are small and serve a useful purpose that would be reused between other agent types.

Behavior trees also offer the flexity to implements tasks that are very small and serve a singular purpose to very large task that are behavior systems in themselves. My patrol task is practically a copy paste of my patrol state and consist of a lot of decisions concerning pathfinding.

The benefit of the a finite state machine is its ease to implement into a solution as it is simple to set up and run a state machine, whereas implementing a behavior tree a lot of work needs to be done before hand to set up and implement the behavior tree.

References

- Py-trees.readthedocs.io. 2020. *Background — Py_Trees 2.0.15 Documentation*. [online] Available at: <https://py-trees.readthedocs.io/en/devel/background.html> [18 June 2020].

- Bevilacqua, F., 2020. *Understanding Steering Behaviors: Collision Avoidance*. [online] Game Development Envato Tuts+. Available at: <https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777> [10 June 2020].

- Sourcemaking.com. 2020. *Design Patterns And Refactoring*. [Available at: <https://sourcemaking.com/design_patterns/state> [8 June 2020].

- Mathur, N., 2020. *Building Your Own Basic Behavior Tree In Unity [Tutorial]* | *Packt Hub*. [online] Packt Hub. Available at: <https://hub.packtpub.com/building-your-own-basic-behavior-tree-tutorial/> [19 June 2020].

- Simpson, C., 2020. *Behavior Trees For AI: How They Work*. Gamasutra.com. Available at: <https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php> [18 June 2020].

- Thompson, J., 2020. *Collision Detection*. Collision Detection. Available at: <http://www.jeffreythompson.org/collision-detection/table_of_contents.php> [12 June 2020].