CRYPTOSHOOTER—MULTI AGENT BASED SECRET COMMUNICATION IN AUGMENTED VIRTUALITY

Submitted By: Sahil Narang, Sarah J Andrabi

PROJECT IDEA

The main idea for the project is to create a pursuit and evade crowd simulation by integrating RVO with Unity Game Engine. The objective is to have a user controlled agent that can 'tag' secondary agents by identifying which of the secondary agents is a 'target' agent and which are not. This step of identifying which characters to tag will be done by using cryptographic techniques which require the user player to align an image token with the corresponding image token of the secondary character.

PROPOSED TIMELINE

- Oct 29: Proposal
- Nov 7th: Integrate RVO2 with Unity & Design scene
- Nov 12th: Motion Model
- Nov 19th : Interaction of simulated agents and main character
- Nov 26th : Catching and Identification
- Dec 6th : Tagging Behavior
- Dec 12th : Final Presentation and Report

APPROACH

Essentially, the project can be divided into the following components:

1. **RVO integration with Unity**
2. **Scene Design in Unity**
3. **Character animation in Unity based on motion models**
4. **Character animation in Unity based on RVO input**
5. **RVO and the simulation loop**
6. **Character interaction and tagging behavior**
7. **Scoring and GameOver**

We briefly describe each of these in this section and then describe each of these in detail in the sections that proceed.

RVO INTEGRATION WITH UNITY

Steps involved in this include:

1. Compile the Library with Unity to gain access to the RVO Simulator
2. Design Scene and Roadmap
3. Add obstacles and agents in the scene and in the Simulator
4. Setting initial preferred velocities for each agent in the Simulator

SCENE DESIGN IN UNITY

Designing the scene in Unity requires placing various game objects in a scene, adding obstacles, figuring out the camera motion and relative positions with respect to the main player, and the bounds of our scene within which all the agents, obstacles and the main player are spawned during run time.

CHARACTER ANIMATION IN UNITY BASED ON MOTION MODELS

Character Animation requires setting up the characters. There are two sets of characters: Main character and Secondary characters. Main character is the user controlled agent and secondary characters are further divided into target characters and non-target characters. The target characters are those the user character has to 'tag' (as will be explained in the next section) and non-target characters are the ones that the user character doesn't tag.

Character Animation requires each character having a motion model that allows them to walk around in the scene and interact with their environment (in the scene) along with their interaction with other characters in the scene.

For the user controlled player, the animations require the character moving around the scene based on the user input, 'isolate' a secondary character, and a 'tagging' animation and the response of both the user controlled character as well as the response of the secondary characters.

CHARACTER ANIMATION IN UNITY BASED ON RVO INPUT

This includes controlling multi-agent behavior based on the RVO model. This component forms the interface between the RVO Simulator and the character animation in Unity i.e. the aforementioned components.

At each time step, the RVO Simulator will take the current position of the simulated agents and the main character and update the goals of the simulated agents such that they always move away from the main character. Consequentially, each agent will call the RVO2 library to set its preferred velocity and update its current velocity so as to avoid collisions with other agents and objects in the scene. The Simulator will then communicate the current velocity to the unity characters for visualization.

RVO AND THE SIMULATION LOOP

The simulation loop consists of the following steps:

1. Get main player's position
2. Update roadmap
3. Set preferred velocities for simulated agents
4. Get collision-free current velocity for each agent using RVO
5. Animate each agent to move with its current velocity
6. Reset roadmap

Each of these is explained in detail in the following sections.

CHARACTER INTERACTION AND TAGGING BEHAVIOR

Character interaction includes allowing a mechanism where the user controlled player figures out if the 'isolated' secondary agent is a *target* or *non-target* agents and finally 'tagging' *(shooting)* the *target* agent.

We plan on implementing this as a basic FSM composed of three events – "Isolate(suspend)" , "Identify" and "Tag". The "Isolated" event fires when the main player is close enough to a secondary agent. Once fired, the secondary agent will be in a suspended state i.e. the agent will not change its position. Once it is suspended, the "Identify" event takes place. At this time the main player brings up its visual secret share and aligns/compares it with the visual share of the secondary target. If the share forms a white box, that implies the secondary agent is a target and the player can shoot it, else the agent is not a target and the main player should not shoot it. The action of 'shooting' is our equivalent of tagging the player.

SCORING AND GAMEOVER

Since the project is designed as a game, scoring is an essential part. Based on which secondary agent was 'shot'/'tagged', the main player—which is the user—gets points. Based on the total points which can either be the maximum or the minimum allowable in the game, the user/main player either wins or loses.

In the following sections we look at each of the aforementioned steps in detail.

RVO INTEGRATION WITH UNITY

1. Compile the Library with Unity to gain access to the RVO Simulator
   We used the RVO2 C# implementation available at http://gamma.cs.unc.edu/RVO2/. We ran into some issues while integrating the library to work with unity:
   a. Multi-threading in Unity
      RVO2 natively supports multithreaded. However, synchronizing multiple threads with Unity is non trivial. Hence, we converted the parallel code to sequential. Given the complexity of the scene, we believe the game should still be interactive.
   b. Singleton

The C# version of RVO2 is not ideal since it uses a singleton for its main class “Simulator”. This allows others objects to communicate with the simulator. It took us a while to work around this constraint but we have it working now.

2. Design Scene and Roadmap
   The scene is designed as a plan with 4 square obstacles in the center with passages between them. The roadmap is constructed by performing visibility queries on manually identified waypoints and the edges of obstacles. Once constructed, the links in the roadmap graph indicate paths that guarantee collision avoidance with respect to obstacles.
3. Add obstacles and agents in the scene and in the Simulator
   Adding and processing the obstacles in the simulator and initializing the agents.
4. Setting preferred velocities for each agent in the Simulator
   Each agent can choose from one of 4 goals, each placed at the edge of the scene. Given the position of the main agent and its current velocity, each agent selects the furthest goal position in the opposite direction. It then queries the roadmap to find a valid path and sets its preferred velocity accordingly.
5. Computing and Communicating Current Velocity
   As a basic setup, we have built a framework than creates spheres with predefined goals and initial positions. Hence, their preferred velocities are constant throughout the simulation. The simulator then calls RVO2 at each time step to compute a feasible current velocity. For example, the following figure is a snapshot of two diametrically opposite spheres exchanging positions.
   Once, the first 4 steps are complete and working, we can plug them into the framework to obtain a full simulator with spheres and then integrate that with the Character Animation framework.

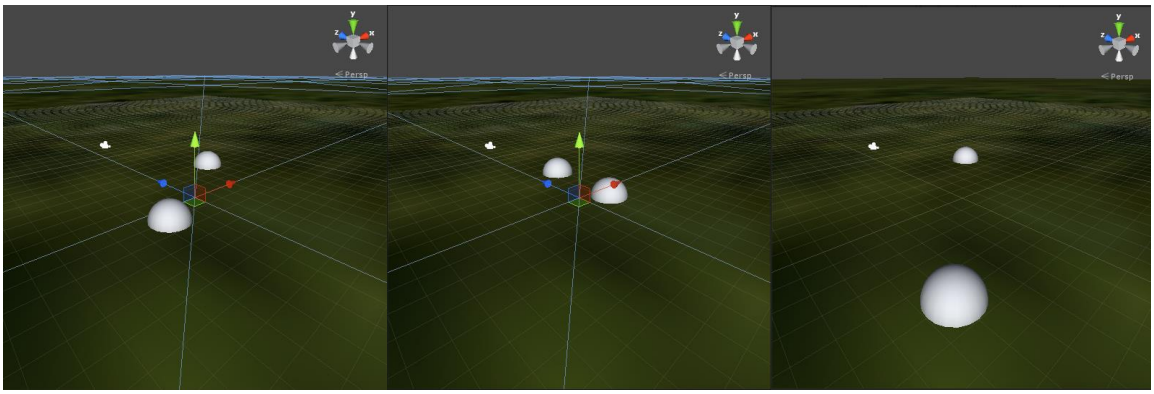


Figure 1. Spheres exchanging positions using RVO2

SCENE DESIGN

SCENE DESIGN IN UNITY

Our scene is designed as a medieval village and consists of various scene elements like buildings, fountains, pillar monuments, wells, carts etc. All the scene elements are placed within the bounds of our game area where the agents and main player will be spawned once the game starts running.

Figure 2. A Screenshot of our scene in Unity

Figure 3 shows our main player game object and Figure 4 shows the secondary agent with a visual tag share hovering at the top of its head. We'll talk about the visual shares later.

Figure 3. Main Player Model

Figure 4. Secondary Agent Model

SCENE ELEMENTS AS OBSTACLES IN RVO

All the scene elements mentioned above are added as obstacles in RVO. Since RVO will be used by the secondary agents to do collision avoidance. These objects also have appropriate colliders (based on object shape) associated with them, to prevent our main player walking through them. This is done since RVO doesn't control the motion of the main player, which is a Unity game object. Hence adding colliders to the other objects as well as our main player, prevents any undesirable behavior.

CHARACTER ANIMATION

MOTION MODELS AND ANIMATIONS FOR THE MAIN PLAYER AND THE SECONDARY AGENTS

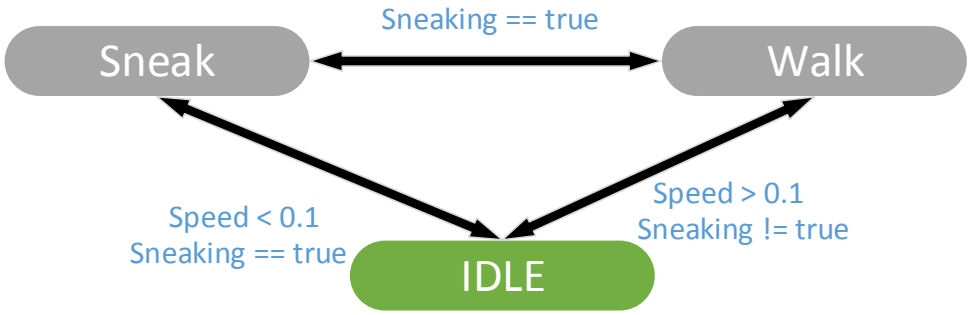
We found the motion models for the player and agents that enable them to walk around realistically. These motion models were part of one of the Unity Tutorials and we are using them for our animations.

The different poses that we are using include the following—Main Player: Idle, Walk, Sneak, Raise Hand, Lower Hand, Move Hand to imply Tagging

Secondary Agents: Idle, Locomotion and Die. Locomotion consists of different states like walking, running, turning left and right at different angles.

ANIMATION STATE MACHINES

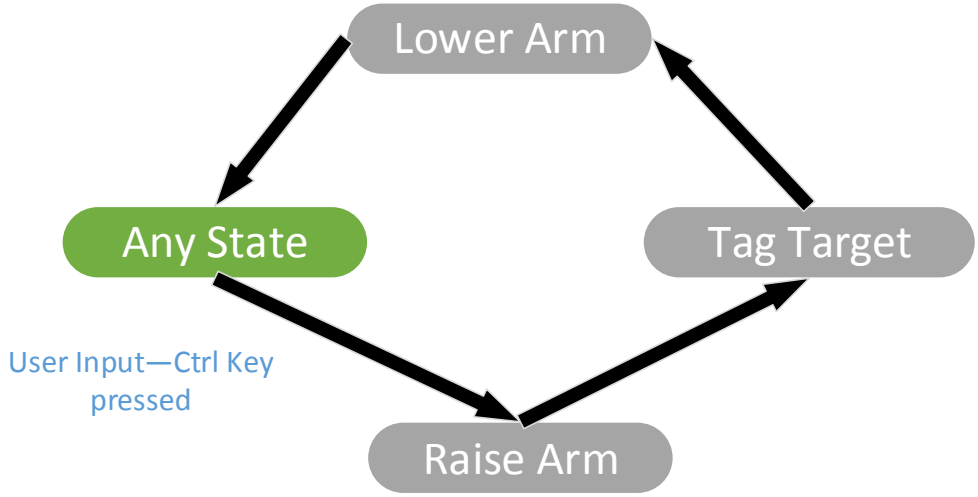
The following state machines (SM) show the logic between going from one state to another for the Main characters as well as the Secondary characters, along with the conditions that cause that transition to the next state. The green box represents the starting state.



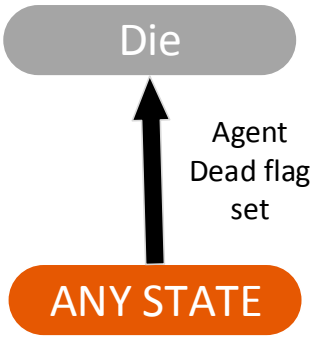
SM 1: Main character state transitions

SM1 shows that our main character will start off in the idle state till the user moves the character and the main character will start walking. If the sneaking flag is true i.e. the user sees that the Main Character is close by then the user can make the Main character sneak up to a secondary character to isolate it.

SM 2 shows the transitions for the actions that the main player will take to indicate that it is taking the action to 'Tag' an agent i.e. raising his arm, a small recoil to indicate the secondary agent has been tagged/shot and then lowering the arm afterwards. Once the main player shoots the agent, the agent dies and its death animation is played, as shown in SM3.

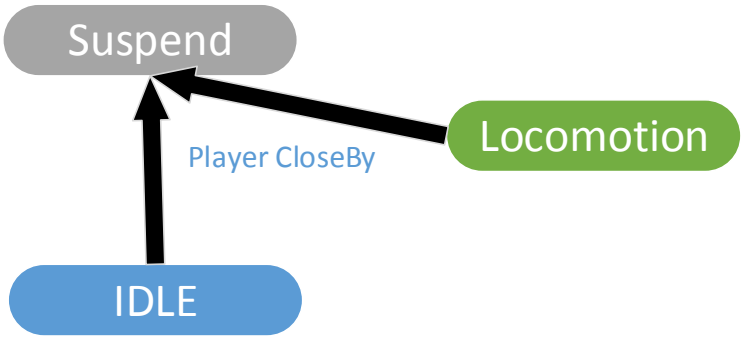


SM 2: Main character state transitions upon identifying target player to tag

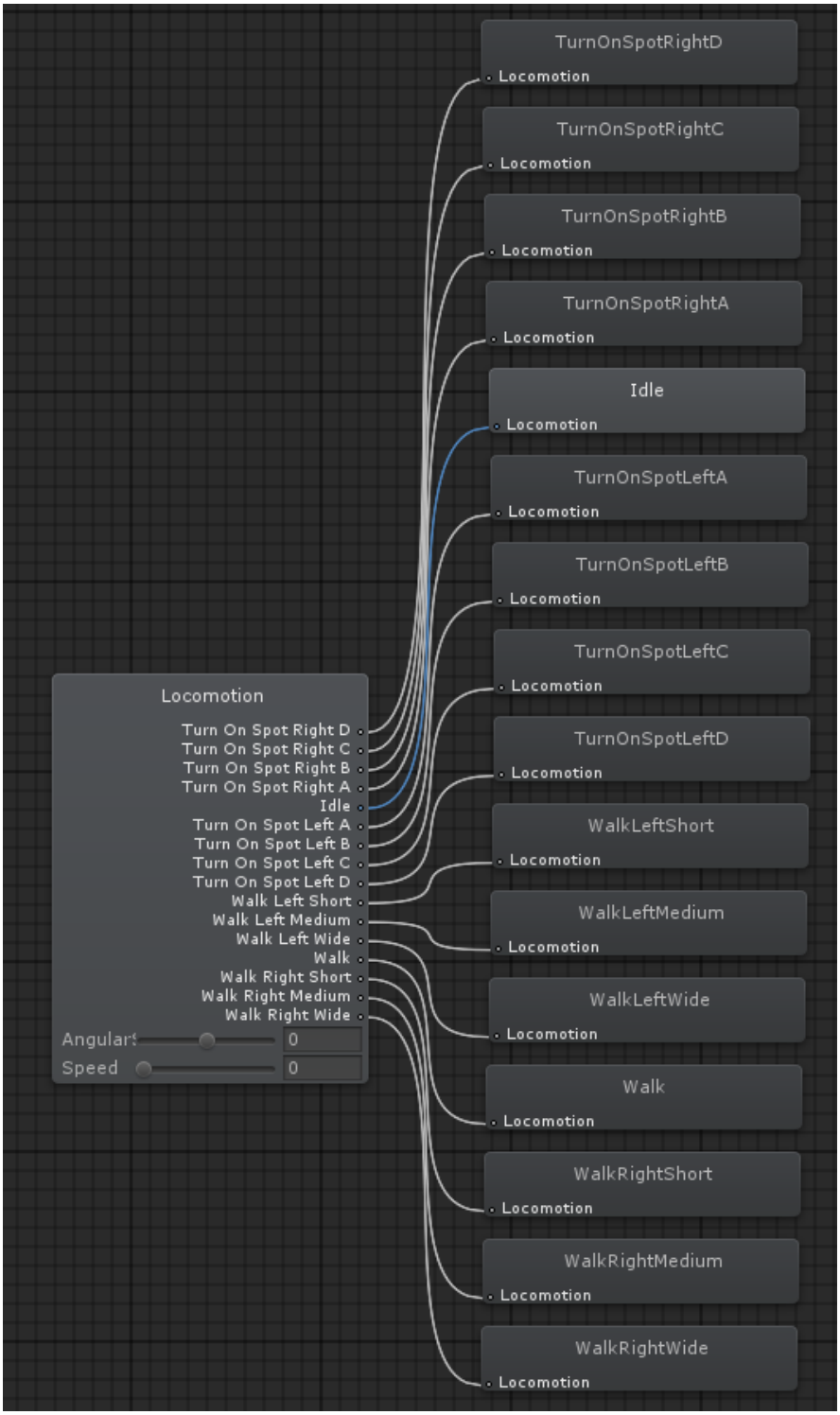


SM 3: Secondary player dying state transitions

SM4 shows the states for the secondary agents. Locomotion has different animations based on whether the character will walk, run or turn, as shown in SM5. If the main player is close enough to the secondary agents then, they go into suspend state and no velocity input from RVO is received. The animation for this state is the same as the Idle state.



SM4: Secondary Character Animation states



SM5: Locomotion states of the secondary character

LOGIC FOR TRANSITIONING BETWEEN ANIMATIONS

Given the many possible locomotion states as shown in SM5, Unity's animation engine (Mecanim) needs to be able to figure out which animation to play. Hence for the same we create a 'Blend Tree' in Mecanim and give it all the possible locomotion animations. Blend Trees are used for allowing multiple animations to be blended smoothly by incorporating parts of them all, to varying degrees. The amount that each of the motions contributes to the final effect is controlled using a *blending parameter*, which is just one of the numeric animation parameters associated with the Animator Controller.

As can be seen in SM5, there are two parameters—speed and angular speed, that are used as blending parameters for the locomotion states. We used Unity tutorials to figure out how to create these and use the various motion states and the blending parameters to make the animations smooth.

The velocities that RVO provides to a single agent are used to provide the blending speed and angular speed for that agent to Mecanim—which then figures out the appropriate animation to play for that agent.

We take the desired velocity as being the one that RVO provides and then use that and the current velocity vector to figure out an appropriate speed value to provide to Mecanim. Using simple vector math we calculate the projection vector based on the two velocity vectors and use the magnitude of that vector, as shown in Figure 5. This will tell Mecanim how much of the new speed to add to the current speed and what animation to play based on that. This is especially useful for cases in which the forward direction is different than the direction of the desired motion, as it prevents the agents from going at a really high speed in the wrong direction. This also makes turning easier and more realistic.

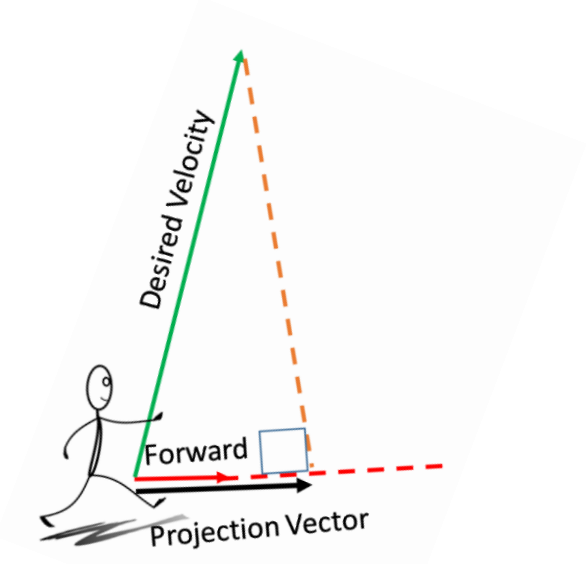


Figure 5. Determining appropriate next speed using current and desired velocity for the Mecanim

Now that we have the speed, we also need to figure out the angular speed as the agent is turning. In order to do that we consider the direction in which agent is moving as the desired direction vector, find the angle between the two by taking the cross product of the two and determine whether it is positive or not. We then use this angle to figure out if desired velocity is to the left or right of the forward vector and turn accordingly. If desired direction vector is to the right of current forward vector the cross product will be positive else it will be negative. Figure 6 shows how this is done. We use this angle to calculate the angular speed and provide that value to Mecanim which then chooses the appropriate animation. However, if this angle is too small, there is a chance that the agent will turn too much and hence we account for that case we don't calculate the angle but make it face the direction of the desired velocity.

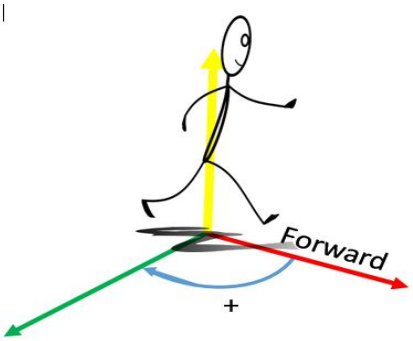


Figure 6. Using Vectors to find the direction of turning

This provides a seamless integration of the RVO velocities for agents and the animations used in Mecanim. In the next section we describe how RVO provides the velocity feedbacks and the simulation loop

RVO AND THE SIMULATION LOOP

GET MAIN PLAYERS POSITION

The first step is to feed the main players position into RVO with a velocity set to zero in order to ensure the rvo agents plan around it.

UPDATE ROADMAP

Update the roadmap by inflating the weights of the edges such that edges closer to the main player have a higher weight than edges far away. This is done by computing the distance to the main from each edge and adding the inverse of the distance to the weight of the edge (ensuring no division by zero).

SET PREFERRED VELOCITIES

This is where the agents are programmed to “evade” the main player. Depending on the position of the main player and its orientation, they pick a start node and goal node and query the roadmap using a Dijkstra algorithm to find the shortest path. However, this assumes the agent is not suspended or killed.

- Do Nothing
    - If suspended OR tagged
- Set Preferred Velocity
    - If player is visible AND within range AND heading towards the agent
        - direction = player’s orientation
        - Start node = closest visible node in direction
        - Goal node = farthest visible node in direction
        - Path = getPath(roadmap, start node, Goal node)
    - Else
        - Continue if not at Goal node
        - Else
            - Start node = Goal node
            - Goal node = random goal
            - Path = getPath(roadmap, start node, Goal node)

GET CURRENT VELOCITY

Once, the preferred velocity is set for each agent, the next step is to query RVO2 to obtain the collision free velocity for the next timestep. For now, the time step is set to unity’s Time.deltaTime which represents the time since the last time the update() function was called.

ANIMATE AGENTS

Each agent is instructed to walk with its current velocity. This involves determine an appropriate speed based on its last velocity such that the degree of turning is controlled. This speed is less than or equal to the RVO speed. We also calculate the angular speed and map them to the mecanim’s animation controls to determine the appropriate animation.

RESET ROADMAP

Finally, we reset the roadmap to its original weight that is based simply on the distance between two nodes in the graph.

CHARACTER INTERACTION AND TAGGING

SUSPEND PHASE

We associate a sphere collider with the main player's character. If any agent enters the collider, its state is set to "suspended". This implies that the agent is immobilized and RVO is no longer queried to obtain its current velocity. During the stage the consequent steps of the player-agent interaction may take place. In case the agent is not killed and it leaves the player's collider, its state is reset and it continues moving on its original path.

IDENTIFY PHASE

Once a secondary agent has entered suspend phase and is no longer moving the main player can identify whether the agent is a target or a non-target. This is done by the use of visual cryptography. Visual cryptography is a cryptographic technique which allows visual information (pictures, text, etc.) to be encrypted in such a way that decryption becomes a mechanical operation that does not require a computer[1]. Traditionally visual cryptographic is carried out by taking an image and splitting it into two components. Each component image, has 2 pairs of pixels for every pixel belonging to one pixel of the original image. The original image is recovered by overlapping the two shares and then the human visual system XORs the two component images. As an example, Figure 7 shows two blocks generated for one pixel of the original image. If we overlap share 1 with share 1, we get a partial black and white block. On overlapping share 1 and 2, we get a full white block.

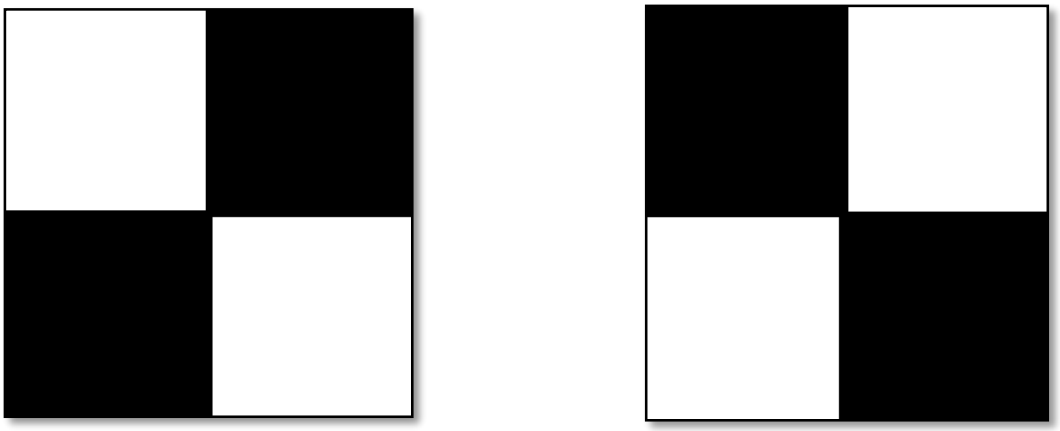


Figure 7. Visual cryptographic shares 1 and 2

This technique has traditionally, been realized through the use of printed transparencies, and with white regions represented by transparent blocks. Our project uses this in an Augmented Virtuality environment.

1—Source Wikipedia

We use this to identify secondary agents as targets and non-targets. Each secondary agent is randomly assigned one of the two visual shares, as shown in Figure 7. The main player also get one of the two visual shares. Target agents are those for which the alignment of the visual shares of the main player and the secondary agent results in a full white square, and for non-target agents it is represented by a partial black and white square. The visual shares used in the scene are such that black regions are transparent and white regions are opaque. Note that we are only using simple shares to make the game easier to play. As a future goal, another level could be added to the game with more complex image shares, which reveal words like 'Tag' or 'No-Tag'.

Each secondary agent's visual share hovers over the agent at all times. The main player's share can be made visible by pressing the left-shift key and can be moved around with the mouse pointer. To hide the player's share, the user needs to press the right-shift key.

Once the player has identified the target agent, he can go ahead to the next phase.

TAG/SHOOT PHASE

After having identified the target agents, the player has to tag them. In this game this is achieved by shooting the player. The player shoots by pressing left-ctrl key, which fires the gun that the player has. A laser beam is fired from the gun by casting the ray in the direction of the target agent. If the ray hits the agent, the player gets points or loses points according to the type of agent he hit(target or non-target) and the agent which is shot dies. If the ray hits no object then it just dies out and no damage occurs. If the ray hits any other object in the game, nothing happens.

SCORING AND GAMEOVER

As described in the previous sections as the player identifies the target agents and shoots (tags) them. When the player shoots a target agent he gets 10 points, on shooting a non-target agent the player loses 10 points. On shooting a target agent the screen flashes green to indicate that a target secondary agent has been shot. On the other hand, on shooting a non-target secondary agent the screen flashes red.

The game begins with the player having some points and the player can gain points to win the game with the maximum collectible points. The player loses the game by losing points and going below a minimum point tally. Figure 8 shows the scoring element that is displayed at the bottom left of the screen to indicate the current score.

Figure 8. Score Element

STATUS AND FUTURE WORK

We achieved all the goals outlined in our project proposal.

Future work includes making the visual cryptography shares more complex to increase the complexity of the game, and integration with Oculus. More scene elements can also be added to make the scene and collision avoidance task more challenging. This project idea can be used in a multi-player gaming environment with each player requiring tags to figure out some shared secret.