Test 2, CPSC 4160/6160 – Spring 2021, due on 4/19, 11:59pm. Submit through Canvas, questions can be posted to Piazza, public or private. Do not disclose any answers on Piazza.

*Answer the following* ***in the space that is given****. You may not use extra space. I recommend you write in a draft document and only fill in the pdf when you are complete. Write your own words, do not quote or (heavily) reference outside sources – integrate your findings and write the answer on your own.*

Each question is 8pts each, totaling 40pts.

Q1) Your employer is developing a side-scroller game that uses parallax to give the sense of depth in the background of the game. In your own words, explain why parallax indicates depth. When considering using multiple layers for parallax in the game, how does this benefit (or hinder) the game’s visuals *and* performance – in contrast to the use of a single layer, or not using the effect at all? Finally, to make the layers appear to have a specific depth difference (e.g. trees at 50ft, mountains at 1000ft), how do you calculate the different speed of the layers? Be specific.

**When I read the word depth, the definition and visualization that comes to my mind is *distance* from the player or camera, or some sort of separation of scenery and objects from one another depending on the *distance*. Parallax in a side-scrolling game is the use of layered imagery or multiple images with alpha values that simulate the effect of *distance* between objects and scenery when moving. When using parallax in a side-scroller, a developer can layer two or more images as a background but paint one on top of the other and make the furthest object back smaller, creating the illusion for the human eye that the 2D world the eye is seeing has depth and *distance* to its background. Utilizing multiple layers for parallax in a game can exponentially increase the visual integrity of the game. However, creating many layers that move at different speeds or grow at different rates can have a negative impact on a game engine that is trying to process everything at once, therefore taking a poll on the performance. A developer using multiple layers for parallax with, for example, the painter algorithm, has the freedom to create more immersion for the player. They can add separate layers for clouds of different sizes or make mountains that are further away (smaller) move slower. This can drastically increase the fidelity of the game’s scenery and immersion but with so many moving parts can cause framerate to drop or graphical glitches more likely to occur. If a developer were to use a single layer for parallax, they could achieve a similar goal, but would struggle to separate pieces of the background, have different speeds for different distanced objects, and create immersion. A developer choosing to use no parallax effect on their background is sacrificing the effect of distancing and immersion completely and is choosing to go for a more old-school, arcade-style side-scroller that will most likely focus on gameplay. A side-scrolling game with a single layer of parallax or no layers of parallax will naturally focus more on performance than visuals in the background. Finally, to have layers appear to have a specific depth, you should first give each layer a default velocity respective of the distance you want it in the background. Next, access the speed that the character is moving at and use that number as a multiplier on top of each layer’s velocity. This would accurately speed each layer up depending on the distance you set as a default and the character’s movement speed. For example, trees that are closer to the character (50 ft) could have a default velocity of 10, while mountains (1000 ft) could have a default velocity of 2. Then, using a character’s movement velocity of 6, increase or decrease each background layer’s movement velocity accordingly and you have accurate parallax!**

Q2) Collisions. We discussed two primary methods for doing an intersection check for collisions. The first being the box collider and the second being the circle/distance collider. For a generic scene with ***n*** objects all capable of colliding against each other, provide a performance evaluation comparison of the two techniques. This evaluation should include an estimate of timing for collisions (response resolution can be left out for this evaluation). What are the considerations that are needed in deciding which of the two techniques to choose, give at least two criteria. Finally, what would you recommend for the final game you have in mind and why, e.g. what additional factors helped you make this decision?

**In a generic scene with n objects that are all capable of collision with one another, the choice of collision detection methods can have an impact on the performance of the system both with calculation intensity and with collision timings. Using box colliders for intersection checks will prove to be less intensive on the performance of the system, as it utilizes only greater than and less than checks inside “if” statements for x and y values on the corners of each box. The circle/distance collider uses radii of each circle/distance collider and compares the distances of both to determine if the edges of the circles are colliding with one another. This method utilizes comparisons like greater than and less than inside “if” statements, but also uses the performance-hitting calculation for the radii distances using a square root formula. After doing some research and some testing of my own, the collision timings for each method showed interesting results. Although more performance-intensive, the circle/distance collider is faster at detecting collisions between objects than the box collider by an average of 50 milliseconds, but the box collider is faster at returning an answer when there is no collision between objects. This means that if a game relies on detecting collisions often, going with a circle/distance collider is the better option for timing, while if a game is not having objects collide often, the box collider method will return a faster time for calculations. The reason for this is the way box colliders are structured in their collision detection. Box colliders are constantly checking two corners of the box on both the x axis and the y axis. To have a collision, both the x axis and y axis must have one of the same values, meaning an edge of the boxes is touching. When optimized, box collision code can check each individual axis’s values and return “false” as soon as it realizes that one axis is not touching instead of processing all four values and checking for two “trues.” Circle/distance collision uses a simple method of one calculation for distance and comparing the two values, making each distance collision calculation a similar, fast speed. The criteria for choosing which technique to use are the shapes of the objects or sprites that require collision detection, the performance intensity the game requires/can handle, and the accuracy of corners on a hitbox. For instance, if a sprite is a round object and fits nicely into a circle shape, the circle/distance collider will most likely be the most beneficial to implement for that sprite. If a developer wishes to utilize the corners of a box, each corner can be calculated in real-time using the box collider method, which also yields higher performance for the system. For the final game I am working on, I think box colliders would be more impactful for the system and provide more accuracy in hit detection. The final game I am working on includes the use of guns and sprites that are not circular shaped. Having box colliders for the player, enemies, the bullets from the gun, and any attack enemies may throw will provide consistency among all colliders and fit the form of each object better.**

Q3) Particles – design a particle system that animates **rainfall** (for atmospheric effects in a game). What are the considerations for this particular game effect? What are each particles’ parameters and how should they be emitted. Give some detail. How should the water droplets interact with the objects in the scene and why? How will you create the visuals for this effect? How could you use randomness in the effect, give two examples.

**My particle system for rainfall would start in a game that is set in an outside environment that has one or two layers of clouds in the sky (preferably dark clouds to fit the aesthetic of a storm). For this particular effect in-game, the developer would need to consider the lifetime of each particle, its x-y location on the screen, and the velocity at which it will “fall” from the sky. Since rain is essentially droplets of water, each particle will be a tiny size, preferably something extremely small that does not look like much alone but in large groups creates a denser effect. For this size, I’m thinking each drop starts at 2pixels x 4 pixels for a realistic rain drop shape and if this proves to be too small, I may bump it up towards 5 pixels x 10 pixels. With the aesthetics in mind, it would be nice to use an actual image for the drops but if this proves difficult a simple dark blue rectangle will suffice. The draw function for each particle will render the rain drop at its current location while the update function will increase each particle’s y-value by a factor of 3 per frame and simultaneously decrease the lifetime count. The maximum lifetime for a raindrop will be 5 seconds, as that seems more than enough time to traverse the entire screen, or until the raindrop collides with another object in the game. The particles will be emitted by many particle emitters located along the bottom y-value of the cloud layer to simulate real rain coming from storm clouds. The water droplets will have tiny box colliders that will interact with anything in the level that can act as a barrier for water falling. A floor, the roof of a house, a tree in the foreground, all of these objects will collide with raindrops and cause the drop to “explode” into three tiny pieces and disappear as a normal raindrop would do when it hits the ground. The visuals for rainfall could easily be designed in photoshop to create the size and shape of the raindrop that I desire, as well as the 2 to 3 frame particle explosions upon contact with an object. As for randomness, there are quite a few ways I can think of to implement this and keep realistic rainfall consistent. My first idea is to have one large particle emitter that stretches across the entire layer of clouds that is currently present on screen and have it emit particles with a random x-value selector for the location. Each frame, the emitter will run a rand() function in its updater and emit a new particle at the random x location. Another form of randomness could be to have the velocity of each particle be random between a set range of values. One particle may emit with a velocity of 10 while the next will emit with a velocity of 2. This, paired with the random location emitter, could create extremely realistic rainfall. One last form of randomness that I thought would be cool to implement is the randomness of the layer that the particle is emitted from. For instance, there are two layers of clouds, one further back than the other and smaller, and there are also two layers of emitters paired with each layer of clouds. The particle emitter function may randomly choose which layer to spawn a particle in and then size that particle appropriately to the layer. This could create the effect of both background and foreground rain that will move slower/faster and look smaller/larger.**

Q4) Game AI. We looked at two primary types of non-player characters (NPCs) AI, a force field method and A\* path search. What are the benefits and downfalls of each? What are the types of games that make better use of each? Describe a setting where you might use both tools in a single game.

**Non-player character AI can appear to be very sophisticated but actually use simple mechanics for maneuvering a world or getting to a goal. The two primary types of non-player character AI that were talked about, force field method and A\* path search method, operate uniquely and with benefits for different game types for each. The force field method is an excellent choice for non-player characters that need to reach a goal or endpoint that will not have a lot of interaction or blockages on the way to its goal. The force field non-player characters are great choices for implementations of crowds or “flocks” and can utilize miniature force fields amongst each other to keep a balance and not override other character’s space. Alongside this, force fields are the perfect choice for very simply moving a non-player character around an obstacle on the way to its goal. It will receive the force from the object and be pushed to the distance it needs to be pushed while still have the shortest path to its goal lined up. Where force fields start to taper off in usefulness is where A\* path search starts to shine: crowded, obstacle-ridden areas. If there are dead ends or a lot of obstacles to maneuver, force fields may lead a non-player character into a loop of getting pushed around or get it stuck in an area where it has no escape. The A\* path search method will allow a non-player character to make a smart traversal of the environment it is thrown into and can help it maneuver an environment that is not very straightforward. It uses a cost-benefit technique to figure out which path is the quickest to the goal and runs similarly to an automatic maze-solving algorithm. The non-player character, before moving, will determine the most efficient route to get to its goal, thereby minimizing its chance of getting stuck in a corner like the force field method. The problem with A\* path search is that it is a breadth-first search and can be more intensive on the performance. It is slightly less efficient than simply telling an object it needs to get to a certain point and sending it off like with the force field method, though the end result will be more calculated and more efficient. The games best suited for force field method AI would seem to be the games with a more guided storyline that uses NPC’s to further the plot instead of being a part of the actual gameplay. Having an NPC wander on screen towards a set goal or having an interactable NPC that walks between two set points would be best for the force field method. The A\* path search would be more fitting for a game requiring NPC’s to navigate a more complex pathway and be more a part of the gameplay. An adventure game with a companion or a dungeon crawler with mobile NPC’s that require precise, real-time movements would be necessary for this type of method. A setting that I could see both the force field and A\* path search methods being implemented would some sort of roaming world shooter that has multiple enemy types of different difficulties. One type of enemy AI will traverse on a set path and move around based on its force field interactions, while the more difficult (boss-like if you will) enemy NPC’s will have an A\* path search algorithm lay out its most efficient pathway in the current environment for it to move inside. A large part of having both types in one game would be an intelligence factor. The “dumber” AI will use force field and simply respond to the environment while the “smarter” AI will use A\* path search to determine which path is most efficient to beat the character and avoid damage.**

Q5) Entity Component Systems (ECS). Define the programming pattern of ECS. Also, state how it applies to video games. What games are bested served by the ECS? What is the savings and the limitations for utilizing this in 2D game engines?

**Entity Component Systems define your game engine in a way that every object currently in the game is listed as an entity and can be modified, moved, or mutated at runtime. Every entity, or object, has an ID that has its own properties and data that go along with it and can be manipulated by the game engine by other functions or manipulations. Entity Component Systems will remove the constraints of having objects, properties, and functions all packed into one entity and will instead separate everything out, creating a freedom of choice of what you want them to do, how you want each entity to be controlled, and what properties you want to give them.**

**Entity Component Systems are best implemented in video games that can use an inheritance structure and have many entities or objects that will be relatively the same and require similar manipulations, such as simulation games where the player creates armies of hundreds of the same entity or create buildings with specific properties that benefit the player. The ECS programming pattern can simplify a complex inheritance tree into an easy-to-understand form where each entity has an ID that links it to specific properties and tells the game what it can and can’t do, and how it can be changed instead of solely relying on only what is has inherited from the class before it.**

**In summary, Entity Component Systems are best used in games that will have a large number of entities or objects present in the game at the current time or in the future. It simplifies the inheritances of any entity that may spawn in the game and require properties to define its limitations. A shooting game that requires a player be controlled with specific attributes and different types of enemies that all need the same properties would work well in an ECS system. The player will have the entity ID that links it to its ability to be moved by the player while the enemies will be linked through the Enemy ID that mean it has some form of AI and collision detection. Roguelikes and simulation-based games with uniquely identifiable ID’d entities would also be good fits for the Entity Component System design pattern.**

**Entity Component Systems can be extremely useful when implementing large-scale, complex games that need entities and objects with similar elements. The ECS method focuses on decoupling objects into separated elements that identify entities and adjusts them to the problems at hand and give them the properties they need. The main positive of Entity Component Systems is that it allows the developer to manage all of the data of the entities in the game in more flexible ways by separating every attribute that can then be managed individually for more precise results while simultaneously affecting a large amount of entities in the desired way. One of the drawbacks of Entity Component Systems is that attempts to introduce new systems or methods in between the already-existing ECS methods can potentially break the functionality of anything already in place. New methods and interactions will not play nice with existing ECS methods. Also, because ECS allows for developers to implement many small systems which can impact a large number of entities, debugging any issues can potentially be very complex and hard to find/fix.**