Game Engine Design 2

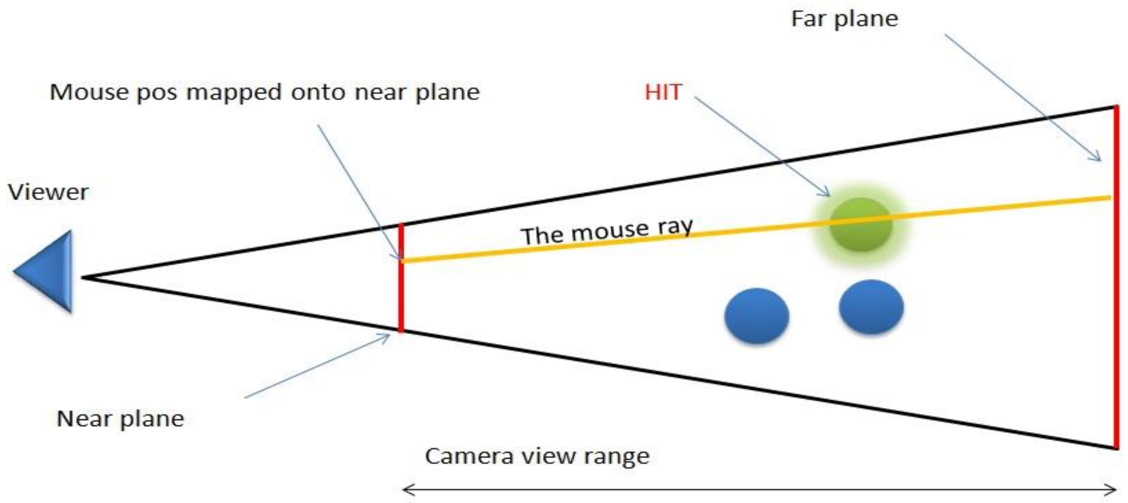
Week Eleven: Mouse Picking

MOUSE PICKING

Ray/OBB Collision and Collision Handling

Mouse Picking

- Collision detection with the mouse is extremely useful to have within an engine
- Mouse picking is done by testing if a ray cast from where the camera is located collides with any of the bounding boxes in the scene
- This will be done using Ray/OBB collision
  - Use this method since the bounding boxes we created are oriented because they save a transform



Rays

- The first step is to create a Ray struct that will be used in the collision detection
- Inside the Math folder create a new item called Ray.h
- In it include glm.hpp and the BoundingBox header file
- The Ray struct should have a vec3 representing the Ray’s origin, a vec3 representing its direction and a float for its intersection distance
- It should have an empty constructor that defaults all of the variables to 0
- The Ray struct should also have a constructor that takes in the Ray’s origin and direction as parameters and sets its variables accordingly
- Create an assignment operator overload for the equals (=) sign that assigns the current ray’s origin and direction to the other ray’s origin and direction
- Finally, create a bool function to check if the Ray is colliding with a BoundingBox
  - For now, have the function set the intersection distance to -1 and return true

Collision Detection

- Under the Math folder create a new class called CollisionDetection
- This class will need to include the Camera class
- As well, simply declare the Ray and BoundingBox structs
  - This is done to avoid a circular dependency since the Ray struct will need to include and use the CollisionDetection class
  - Do not forget to include the Ray header in the Collision Detection CPP
    - Only need to include Ray because it already includes BoundingBox
- This class will be a fully static class so make sure to get rid of the destructor declaration and delete the constructor so that no other class can make an instance of this class
- Create a static function that takes in the mouse's coordinates, the screen's size and a Camera instance and returns a Ray and converts a screen position to a ray
  - This is done to take the mouse's position and convert it to an origin and direction for the ray
- Create a static bool function that checks if a Ray intersects with a BoundingBox

Converting a Mouse Click to a Ray

- First thing that needs to be done is finding the ray's start and end position in NDC
  - Use NDC since it's much easier
- For the start point of the ray, create a vec4 that has the following values:

1. X: Subtract 0.5 from the mouse's X coordinate divided by the screen's width, then multiply that value by 2
2. Y: The same thing as the X, but using the mouse's Y coordinates and the screen's height
3. Z: Set to -1 since in NDC the near plane starts at -1
4. W: Set to 1

- For the end of the ray, create a similar vec4, but its Z value should be 0 instead of -1

Converting a Mouse Click to a Ray – Cont.

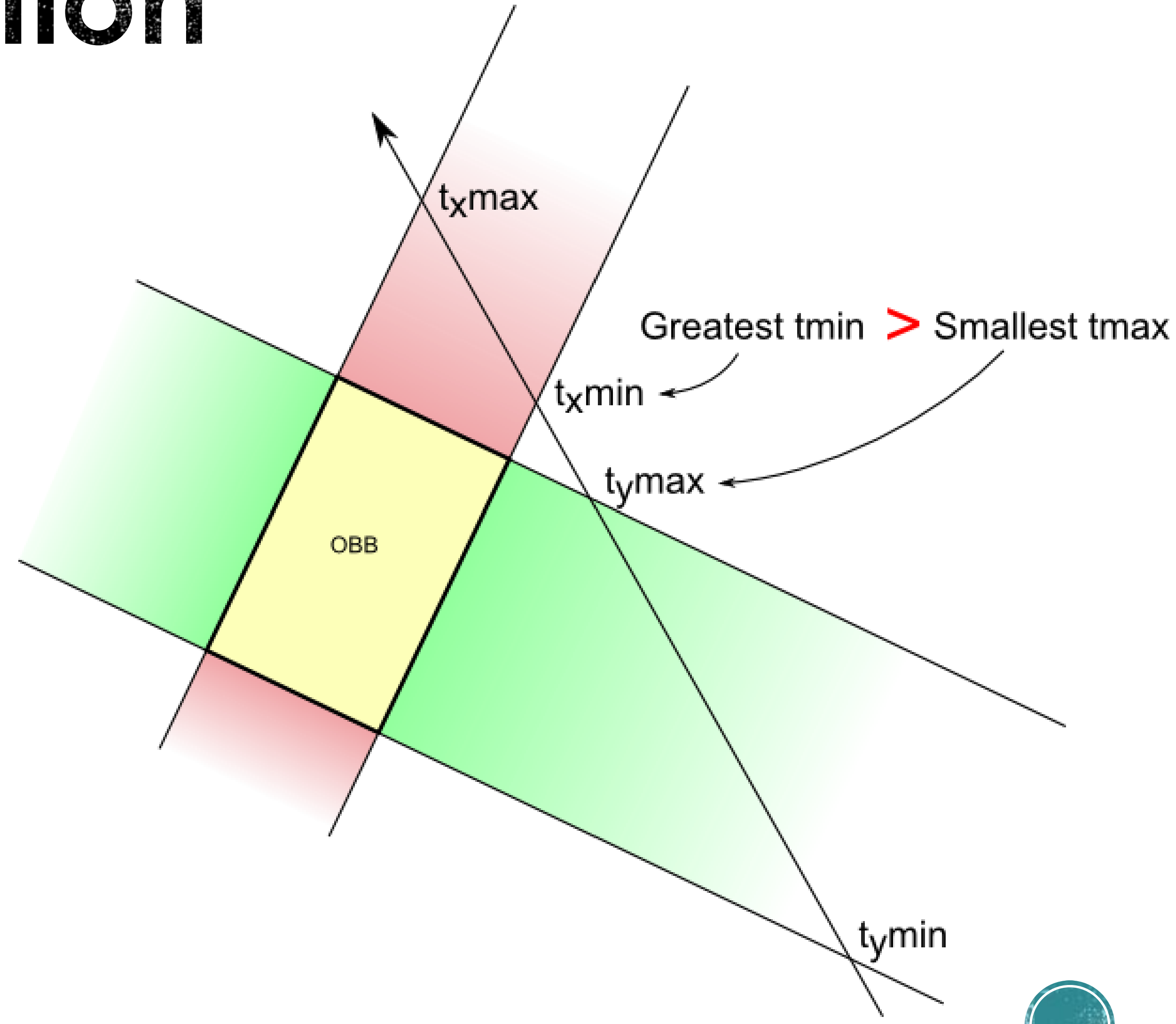
- Next step is to invert the camera’s perspective and view matrices
- Remember that the goal is to go from NDC to world space
  - Invert the perspective matrix to go from NDC to camera space
  - Invert the view matrix to go from camera space to world space
  - Speed up calculations by inverting a matrix made of perspective matrix multiplied by the view matrix
- To find the vec4 representing ray’s starting point in world space, multiply the inverse matrix by the ray’s start in NDC
- Then divide that by its W value
- To find the vec4 representing the ray’s end point in world space, perform the same operation, but with the ray’s end in NDC

Converting a Mouse Click to a Ray – Cont.

- Now to find the ray’s direction in world space is simple – subtract it’s start from its end
  - Same as with a vector, always subtract where you started from where you’re going
- Then, make sure to normalize this vector
- Convert a vec4 to a vec3 (what we need for a ray) by simply creating a new glm vec3 and passing in the vec4 as a parameter into it
  - This is one of the advantages of using glm, it has a number of built in functionality
- Finally, return a new Ray that has an origin equal to its world start point and a direction equal to the normalized direction in world space

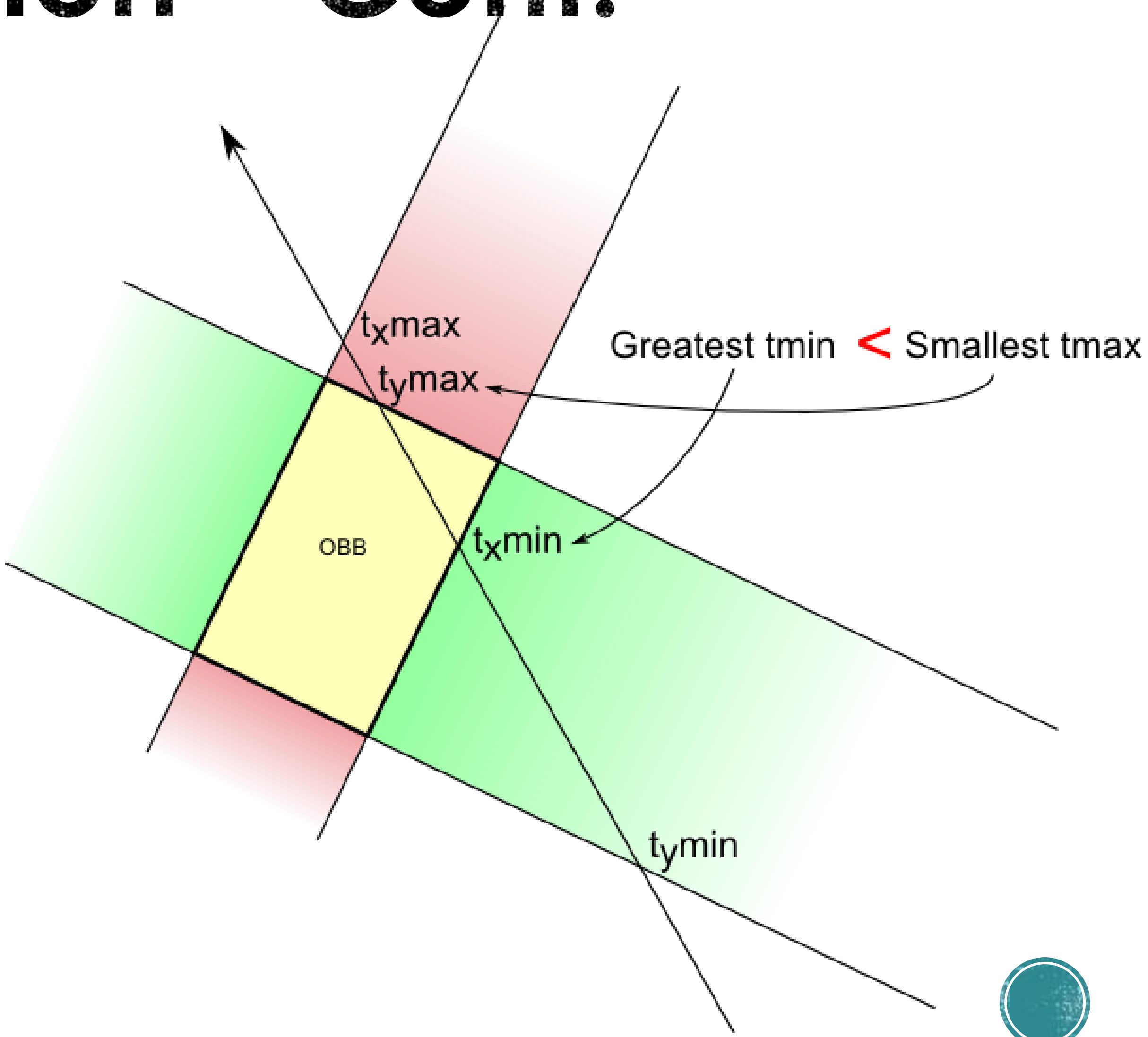
Ray OBB Intersection

- Let's look at this possible ray/OBB collision
- When a ray intersects with the OBB it creates 4 intersections – a minimum (near) one and a maximum (far) one in both the X and Y directions
- Notice that in this potential intersection the ray enters the green (y axis) area, leaves the green area, enters the red area (the x axis) and then leaves the red area
- If the ray does this it means that there was no intersection between the ray and OBB



Ray OBB Intersection – Cont.

- On the other hand, now take a look at this possible ray/OBB intersection
- Notice that in this instance, the ray enters the green area and then enters the red area without leaving the green area
- If this happens, then that means an intersection has occurred



Adding a Function to the Camera

- Before we can start creating code to check for ray/OBB collision, we need to create a function to get the camera's near and far planes
- Instead of creating 2 separate functions, we will create one that returns a vec2
- Create a public vec2 function called GetClippingPlanes
  - Make sure that the function is a const one so that no class can change the near and far planes
- This function should return a vec2 which has an X value of the near plane and a Y value of the far plane

Ray OBB Intersection Detection

- Include the CoreEngine header file in the CPP since we will need access to the engine's camera
- Start the function with defining some variables we will need

1. A mat4 to hold the model matrix, which is the bounding box's transform
2. A vec3 for the ray's origin and another for the ray's direction
3. A vec3 to hold the aabb minimum which is the box's minimum vertex
4. Another vec3 to hold the aabb maximum which is the box's maximum vertex
5. A tMin variable and set it to the camera's near plane
   - Remember that to get the camera, we now get it from the engine
6. A tMax variable and set it to the camera's far plane
7. A vec3 to represent the box's world space which is equal to the x, y, z values of the last column in the model matrix
   - Remember that the columns of the matrix start at 0 so the last column has a index of 3
8. A delta variable that is used to calculate the intersection with the planes which is equal to the box's position minus the ray's origin

Ray OBB Intersection Detection – cont.

- To calculate an intersection in the X axis first create a vec3 that saves the box's X axis by accessing the x, y, z values of the model matrix's first column
- Then create a variable (called e) that is the dot of the X axis and the delta
- Create another variable (called f) that is the dot of the ray's direction and the X axis
- There are two cases we need to check for – a standard one and a rare one
- Lets start with the standard one
- First check if the absolute value of the f variable is greater than 0.001
  - This is to prevent dividing by 0 or nearly 0
- If it is then create a t1 variable that is equal to the e variable plus the aabb's min X value all divided by f
- Then create a t2 variable that is equal to the e variable plus the aabb's max X value all divided by f

Ray OBB Intersection Detection – cont.

- Then check to see if t1 is greater than t2
- If it is, that means that they need to be swapped
  - Need to swap them because we want t1 to represent the nearest intersection and the t2 to represent the farthest intersection
- Then check to see if t2 is less than tMax, if it is then set tMax to t2
- Do this because we want tMax to represent the nearest of the two maximum intersection
- After check is t1 is greater than tMin, if it is then set tMin to t1
- Do this because we want t1 to be the farthest of the two minimum intersections
- Finally, if tMax is less than tMin that means no intersection has happened, so return false
  - This is because if tMax is less than tMin that means that what we denoted as “far” is actually closer than what we denoted as “near”

Ray OBB Intersection Detection – cont.

- Next to check the rare case
- The rare case is when the ray is almost parallel to the intersection planes
- If it is, then that means no intersection
- This happens when either of these situations occur:

1. The negative e variable plus the aabb’s minimum x value are greater than 0
2. The negative e variable plus the aabb’s maximum x value are less than 0

- Since no intersection happens in this case, return false

Ray OBB Intersection Detection – cont.

- Now that we saw the calculations for seeing if an intersection on the X axis has occurred, the calculations for the Y and Z axis are almost the same
- The only difference is when creating the Y axis variable, it is the x, y, z values of the model matrix's second column
- Then don't forget to change all the checks following so that it uses the Y value of the vectors
- When creating the Z axis, it is the x, y, z values of the model matrix's third column
- Again, don't forget to then change all the checks to check the Z value of the vectors
- After checking all three axes, set the ray's intersection distance to the tMin value
- Finally, if none of the return false statements were triggered, then that means a collision has occurred, so return true

Connecting the Collision Detection

- Inside of the Ray file, include the CollisionDetection header file so that it can be used
- Will use the Ray/OBB intersection in the ray’s IsColliding function
- First set the ray’s intersection distance to equal -1
  - This is so that if no collision occurred, the variable is still set to a non-zero number
- The IsColliding function should return what the Ray/OBB intersection function return when passed the current ray (i.e. this) and the box that is a parameter of the IsColliding function

Collision Handling

- Now that we have a way to detect collisions, let’s create a way to handle collisions
- Under the Math folder create a new class called CollisionHandler
- This class will need to include: CollisionDetection, GameObject and Ray
- This will be a singleton class so move the destructor and constructor to be private and create a private unique_ptr and default_delete
- Create 2 private vectors of type GameObject pointers called colliders and previousCollisions
- Create a public static function to get the instance of this class
- Create a public function to create the handler, to add a game object to the handler, an update function and a destroy function
  - This update function should take in a vec2 for the mouse’s position and what mouse button was clicked

Collision Handler

- Inside the CPP, include the engine’s header
  - This is done in here to prevent circular dependencies
- Redeclare the static variables that were created in the header (the unique_ptr and the 2 vectors of type GameObject pointer)
- The constructor can be left empty, since we don’t have anything for it to construct
- Inside of the destructor call the OnDestroy function
- Inside the OnDestroy function loop through the colliders vector and set each one to nullptr
  - Only set to nullptr and not delete since it is not the collision handler’s job to delete all the game objects
- Then clear the vector

Collision Handler – Cont.

- The GetInstance function should be set up in the same way as all previous GetInstance functions were
- Inside of the OnCreate function, make sure that both the colliders and previousCollisions vectors are cleared
  - This function will be put into more use next week when we create spatial partitioning
- The AddObject function pushes to the colliders vector the GameObject that is passed into it

Collision Handler – Cont.

- The Update function should do several things

1. Create a Ray using the ScreenToWorldRay function and the mouse's position that is passed to the update function
2. Loop through all the objects in the colliders vector and see if the ray is colliding with the object's bounding box
3. If it is, verify that the ray's intersection distance is the minimum throughout the vector. If it is then the current object is the GameObject that was hit
4. If the hit object exists (i.e. if an object was hit) set the its hit to be true
5. Then loop through all the previous collisions and as long as the object you are looking at does not equal the current hit object and the previously collided object does not equal nullptr, set its hit to be false and set the pointer you are looking at to equal nullptr
6. Finally, clear the previousCollisions vector and as long as the hit object exists, push it to the previousCollisions vector

Update Scene Graph and Game Object

- Inside the SceneGraph header file, include the CollisionHandler header file so that it can be used
- Inside the SceneGraph's AddGameObject function, whenever a game object is added to the map, add the object to the collision handler
- In the previous slide it was mentioned to set the GameObject's hit to true/false, let's create that functionality now
- Inside the GameObject's header file, create a private bool to represent the hit status
- This variable will be defaulted to false in both the GameObject constructors
- Make a public function that takes in the new hit status and what button was used
- This function will set the private hit status variable to the value that is passed in
- Also create a public function to get this hit variable
- For Debugging purposes, you can create a cout line that if the hit status is not true, print out which GameObject was hit using its tag as an identifier

Calling the Collision Handler

- Now that the CollisionHandler is all set up, we need to actually use it
- Inside of the Engine's NotifyOfMouseReleased is where the CollisionHandler's Update function will be called
- This function is called when the mouse is released and not when it is pressed down since in most games, application, software, etc. most mouse actions happen on a button release, not a button press
  - Think about this next time you are playing any game or using any application
- The mouse position that is passed in is a vec2 made up of the X and Y the NotifyOfMouseReleased function has as parameters
- The mouse button that was released is the same one that the NotifyOfMouseReleased function has as a parameter
- Finally, inside the GameScene class, after creating the Camera (which should always be the first thing done in the OnCreate function) call to create the Collision Handler