Robert Wascher
GAM 490
Checklist

Video Demo link: https://www.youtube.com/watch?v=Di6_K8WZl0s

1) Game Data Driven
   a) Input Queue
      i) The input queue is found in InputQueue.cs. For the input queue, I serialize a Queue_Data struct. This struct has an in-sequence number, an out-sequence number, a type, and an object. The object always of type Message. There are 5 different types of messages (all of which are subtypes of Message): ShipInputMsg, UpdateMsg, CollisionMsg, CreateMissileMsg, CreateBombMsg.
   b) Output Queue
      i) I have a single output queue that is found in OutputQueue.cs. All messages and data, that are sent, are sent through here.
   c) Network Communication
      i) The only place data is exchanged outside of the queues is the fields governing the prediction, smoothing, network quality, and packet speed. These fields are stored in the NetworkSession.SessionProperties by the server and retrieved by the client.
   d) Change-list #s: 111808, 111816, 112558, 112561, 112582, 112923, 112999, 113014, 114229, 114596, 114705, 114961, 115234, 115346, 115368, 115423, 115447, 115673, 116307, 116319, 117236, 117399, 117451, 118940, 118957
   e) Summary of Work
      i) The first thing I did for this section was to go through the code and try to find areas that were going to be needed to be converted into Data driven calls. I added comments to these sections in order to find them again. The biggest section that I recognized as needing to be converted to data driven calls was the inputs. I started out using several different structs to handle the data of these calls. I then went about creating the input and output queues. After implementing the queues, I had the output queue send the Queue_ data packets directly to the input queue. This allowed the game to still be played through the queues.

         Soon after this, I realized that it would be better to encapsulate the majority of the work that the struct were doing into message classes. This also allowed for inheritance to reduce the workload in certain situations like having each message type implement the abstract method execute and have each message type pass it's type to it's superclass upon creation.

         At this point, I was handling the inputs well with the queues. However, I knew I needed to do all the processing of the physics world on one CPU and send updates across. This led to the creation of Update and Collision messages. After I got the

Lobby working and connecting, I had the output queue write packets using XNA's PacketWriter class and send them across the wire to the other machine. The other machine's input queue would then read the data with XNA's PacketReader and insert it in the queue. After some work to get the collisions and updates working well, I realized I needed a way to notify the other computer of missile and bomb creation. This led to the creation of CreateMissileMsg and CreateBombMsg. After these were created and working proper, the only work left in this section was fixing certain things that were not working as needed or issues that arose from later section's work.

2) Lobby
   a) Is Lobby working?
      i) The lobby cycles from select screen into game and back again. I create a MenuScreen class that uses the MainMenu.png graphic and has some screen prompt for Creating/Joining Sessions. The prompts are just some text at the top of the screen.
   b) System Link or Live
      i) I use system link exclusively because I had no access to any Xbox Live accounts.
   c) Where is the address of the external machine stored?
      i) The address of the external machine is stored under NetworkSession.RemoteGamers[0]. The NetworkSession field is found in the Game1 class.
   d) Change-list #s: 113488, 113495, 113515, 114229, 114248, 114278, 114573, 114596, 117326, 117335, 117341, 117345, 117356, 117358, 117362, 117363, 117373, 117385, 118816, 118820, 118826, 119076, 119584, 119589, 119592, 119603, 119630, 119632, 119640, 119647, 119657
   e) Summary of Work.
      i) I knew I was going to need some place to encapsulate the majority of the logic for creating and joining a session somewhere as well as the need to create some sort of menu screen. Therefore, the first thing I created was a MenuScreen class. The menu screen class has its own update and draw method. I added a menu state into the gameState enum found within the Game1 class. In Game1's update and draw method, there is a conditional on gameState. If the state is menu state it will call the menu's update and draw method, otherwise it will continue with the games update and draw logic. The menu screen uses the MainMenu.png image as well as some text prompting the user to create or join a session.

         The update method of the menu screen checks for specific inputs (a and b) and calls the appropriate method, createSession() or joinSession(). These methods create a session or join a session already created as well as hooking events in the case that a gamer joins or leaves. These methods update/change the games NetworkSession field called netSession. At this point, when I was ready to try to connect 2 computers using this method, I ran into some trouble because, as I later discovered, I was not updating the NetworkSession. After I got the 2 computer connected after this trouble, I decided

to go on and work on getting the ships moving. I knew I would later need to revisit the lobby in order to add proper cycling.

After getting the game networked and running decently well, I decided it was time to add cycling to the lobby. A good portion of this work was done with updating the text being displayed at various states the game could be in. I also needed to update the SessionEnded and GamerLeft event handlers. After adding some logic to these areas the game cycled back to the menu screen properly upon a session ending or gamer leaving. However, I ran into some trouble with reentering the game. I could reenter the game but game was not playing correctly, especially on the client side. After several hours trying to hunt down this error, I figured out that the static counter used to assign all the ids for game objects was not getting reset upon a game closing. This led to the games on each computer being unable to properly communicate because all the game objects on the game that didn't close had different/higher id numbers than those on the other game. After fixing adding this reset to the SessionEnded and GamerLeft event handlers, I only needed to revisit the lobby for a couple minor tweaks.

3) Moving Ships
   a) What data are you transferring between machines?
      i) I am sending several different types of data during the course of the game. I am sending a message from each player that holds fields pertaining to inputs the player has given. This ShipInputMsg contains a Vector2 for the direction of a linear impulse, a float for the rotating of the ship, and 3 enums for the PlayerID, whether they fired a missile, and whether they dropped a bomb. These inputs allow the host side to process all inputs and do the required processing on the host machine. I also send an UpdateMsg to move the ships on the Client machine. This message contains an enum for player ID, an int for Game object ID, a Vector2 for location, a Vector2 for velocity, a float for rotation, and a float for velocity of rotation. In order to create missile and drop bombs on the Client machine I added a CreateMissileMsg and CreateBombMsg, these message both contain a player id enum and a GameObject id int.
   b) How do you differentiate the local machine (host) and the external machine?
      i) I differentiate the local machine using by storing the LocalNetworkGamer gamer upon creating the NetworkSession. This field has a bool isHost. I use this as the primary distinguisher of machines. If I need to find the external machine, I use NetworkSession.RemoteGamers[0].
   c) Where are Collisions controlled?
      i) Collisions are registered and controlled on the host machine. Upon a collision being registered, a CollisionMsg is sent to the client with the 2 Game Objects ID int as well as a Vector of the location of collision. This allows the client machine to trigger the collision events on its machine as well.

d) Change-list #s: 117302, 117310, 117399, 117416, 117419, 117426, 117442, 117451, 118838, 118842, 118847, 118939, 118940, 118941, 118942, 118943, 118957, 118959, 118961, 118970, 118974, 118976, 118980, 118985, 119561, 119569, 119584, 119591, 119634, 119667

e) Summary of Work

i) The First thing I did when working on this section was to get ships and data moving across the network while just sending message to everybody. This multicast allowed for the game to be played across the network. However, it caused the client side to be behind the host client. This strategy did allow me to see if my how I was using PacketWriter and PacketReader was working properly. From there, I slowly added in logic to make all the state of the game to be controlled by the host machine. I then put conditionals input queue to differentiate ShipInputMsgs from one player vs. that of the other. If the machine was the client machine and received inputs from the host machine player it would immediate add it to the output queue. The host machine processed all input messages it receives. From here, I had the ShipInputMsgs be sent only to the other machine. If the input was from the host machine it would loop through the client machine and return. This was the inputs will be in the same time instance of time on both machines.

Once the inputs were working correctly, I went to work on making sure the update messages were working properly. Most of the bugs were found when working with the multicast of data. However, a few still existed. Because the host is the only machine updating the physics world, only the host creates these messages and only the client machine needs to process them. I, therefore, added in the necessary logic to make sure that this occurred. After the ships were flying correctly, I noticed that no ships or bombs were created on the client machine. This led to the creation of CreateBomb and CreateMissile messages. After fleshing those out and getting them to work properly, I turned my attention on getting the collisions working correctly on the client side. Once working out the kinks that came with the CollisionMsgs, I only needed to revisit this section to tackle a few bugs that arose.

4) Prediction/Estimation

a) Did you complete the prediction/estimation?

i) To throttle the network speed, I have a field FramesPerUpdate that has 3 possible values: 1, 3, 6. This is changed by pressing the L key and controls how many frames must pass before an update packet is read. To throttle network quality and packet loss rate, I have an enum field netQuality that has 3 possible states: poor, good, perfect. This is changed by pressing the K key. Poor state sets the simulated latency to 200 ms and has 20% packet loss. Good state sets the simulated latency to 100 ms and has 10% packet loss. Perfect state has 0 simulated latency and has 0 simulated packet loss.

b) Change-list #s: 119099, 119104, 119117, 119120, 119126, 119127, 119129, 119132, 119682, 119691, 119731, 119735, 119743, 119753, 119766, 119768, 119783, 119788, 119791, 119886, 119897

c) Summary of Work

i) The first thing I did was create a netPredMan class to contain the majority of the logic for prediction and estimation. This class has its own update and draw method that are called within Game1's update and draw method. The draw method just draws text on the screen displaying the Network Quality, Packet speed, and whether prediction and smoothing are on or off. The update sets the network to behave appropriately for the give state. The input controls for these field are taken care of in Game1's checkInput() method. Also in netPredMan's update method is a call that calls each players ship to update for smoothing and prediction if the fields are marked on.

The netPredMan's update method controls how much simulated latency and packet dropping occur in the network session. It also places or retrieves values for these fields stored in NetworkSession.SessionProperties. The host places the values there while the client retrieves them.

Each ship class has methods that control the majority of the prediction and smoothing logic. I added velocity and vRot to the Update packets to allow for prediction to be computed easily. I also added field for oldLocation, oldRotation, newLocation, and newRotation in the game object class that are used as needed for prediction and smoothing. The newLocation and newRotation fields are used only by the smoothing. Each ship has a method that checks whether prediction is toggled and/or smoothing is toggled. It then calls the appropriate updatePrediction() or updateSmoothing method. updatePrediction() adds velocity and vRot to the location and rotation of the ship if an update wasn't received. updateSmoothing() uses lerp functions to linear extrapolate a position/rotation between its location/rotation and the newLocation/newRotation field.