Create – Applications From Ideas
Written Response Submission Template

Please see Assessment Overview and Performance Task Directions for Student for the task directions and recommended word counts.

Program Purpose and Development

2a)

The program was a website designed to be a multiplayer game in which up to four people could join the same game "room," in which the users could see each other's cars. The programming languages involved were JavaScript, HTML, and CSS. We used Node.JS for the server-side code, using the socket.io library to establish websockets for real-time interaction with the client-side code and the express dependency for routing, as well as the Three.js library client-side library to aid with 3D rendering. The intended purpose of the program was to create a fun, multiplayer game involving physical controls (i.e., tilting a smartphone as opposed to pressing keys) that could be played with any modern smartphone. The video demonstrates the basic server functionality (creating, entering, and leaving the game room), and show example gameplay of two players using iPhone controllers and an iMac as the display computer.

2b)

The iterative process for debugging errors involved logging relevant variables, commenting code added since the latest working deployment, and paying close attention to error messages.

One problem I solved independently was that the creation of socket.io (websockets) connections would not always be created before the express (routing) connections, making necessary websocket verification during routing difficult. After identifying the problem by logging variables and realizing that the websocket was sometimes undefined, I tried putting an arbitrary 500ms delay before the verification, but the websocket connection wasn't always created in time on slow Internet connections and long delays ensued on fast connections. I improved this by repeatedly checking for the websocket connection on a 50ms interval, putting some extra strain on the server but ensuring both connections and lessening unnecessary delay.

Another hurdle I overcame independently was figuring how to UV-map the car (wrapping a 2D design over the 3D car shape), a concept I wasn't familiar with. This involved searching documentation and slightly tweaking parameters on demonstrative code to see their effect. I logged the default UV parameters and changing one UV plane at a time, refactoring the UV mapping code into a loop afterwards to remove redundancy.

2c)

One algorithm is the creation of a game room, which happens between the `#createGame` button click and the loading of the game screen on the computer, and is necessary to ensure a unique game id is created and the user is eligible to join.

Clicking the `#createGame` button (in `public/index.html`) begins the algorithm, triggering the event handler below in `public/js/index.js`.

```
/**
  * Create a game when button is clicked
  */
var createGameButton = document.querySelector('#createGame');
createGameButton.addEventListener('click', () => {
  // redirect to page on click
  socket.emit('createNewGame', newGameId => {
    window.location.href = `${window.location.href}game/${newGameId}`;
  });
});
```

This event handler sends out the "`createNewGame`" signal to the server, which is handled by `server.js` event handler (first embedded algorithm) below:

```
// handle when a person creates a new game
socket.on('createNewGame', callback => {
  // make sure user is not already in a game
  if(socket.handshake.session.gameId !== undefined) return;
  // generate random id of five letters
  var gameIdCharacters = 'abcdefghijklmnopqrstuvwxyz';
  var gameId;
  do {
    gameId = '';
    while(gameId.length < 5) {
      gameId += gameIdCharacters.substr(Math.floor(Math.random() *
gameIdCharacters.length), 1);
    }
  } while(Object.keys(rooms).indexOf(gameId) !== -1);
  rooms[gameId] = { host: null, clients: [] };
  callback(gameId);
});
```

This event handler uses two nested loops to randomly create a unique five-character alphabetic game code randomly, which is sent back to the event handler in `public/js/index.js` using a callback function. After receiving the game code, the event handler redirects to the url "/game/(game id)," which is then handled by the routing function in `server.js` shown below (second embedded algorithm):

```
app.get('/game/:gameId', (req, res, next) => {
  // send to game file
  res.sendFile(`${__dirname}/public/game.html`);
  // get gameid parameter
  var gameId = req.params.gameId.toLowerCase();
```

```
  var socket;
  // sync up to socket to join room (keep refreshing until socketId is
updated)
  var syncInterval = setInterval(() => req.session.reload(() => {
    if(req.session.socketId !== undefined && (socket =
io.sockets.sockets[req.session.socketId]) !== undefined) {
      clearInterval(syncInterval);
      // error 1: room does not exist
      if(Object.keys(rooms).indexOf(gameId) === -1) {
        socket.emit('err', `Game room "${gameId}" does not exist.`);
        return;
      }
      // error 2: room has more than four people in it
      if(rooms[gameId].clients.length > 3) {
        socket.emit('err', `Game room "${gameId}" is already full.`);
        return;
      }
      // error 3: user is already in the game
      if(rooms[gameId].clients.find(client => client.sessionId ===
req.session.id) !== undefined || (rooms[gameId].host &&
rooms[gameId].host.sessionId === req.session.id)) {
        socket.emit('err', 'You are already in this game on another tab.');
        return;
      }
      // add gameId to session, session id to game room
      req.session.gameId = gameId;
      // if first person, then host; if not, then client
      if(rooms[gameId].host === null) {
        rooms[gameId].host = {
          sessionId: req.session.id,
          socketId: socket.id
        };
        req.session.host = true;
      } else {
        // create default client
        rooms[gameId].clients.push({
          sessionId: req.session.id,
          socketId: socket.id,
          name: null,
          x: 0,
          y: 0,
          z: 0,
          acceleration: 0,
          speed: 0,
          heading: 0,
          turn: 0
        });
        req.session.host = false;
      }
      req.session.save();
      // join game room
      socket.join(gameId);
      socket.emit('gameId', gameId);
```

```
            io.to(gameId).emit('updateUsers', rooms[gameId].clients.map(client =>
client.name));
            console.log(`A user with socket id ${socket.id} has joined the room
${gameId}.`);
        }
    }), 50);
});
```

This algorithm uses logic (if-statements) to verify that the user can join the game room and determine whether the user is a host or client, accordingly assigning the correct attributes to the server's game room variable, and finally routes the user to the `public/game.html` file.

2d)

```
socket.on('updateUsers', names => {
  /**
    * Position name on top left of correct screen
    */
  var positions;
  switch(names.length) {
    // one person joined: full screen
    case 1:
      positions = [ [ 0, 0 ] ];
      break;
    // two people in the game: side by side
    case 2:
      positions = [ [ 0, 0 ], [ width/2, 0 ] ];
      break;
    // three people in the game: top two side by side, bottom in center
    case 3:
      positions = [ [ 0, 0 ], [ width/2, 0 ], [ width/4, height/2 ] ];
      break;
    // four people in the game: top two side by side, bottom two side by side
    case 4:
      positions = [ [ 0, 0 ], [ width/2, 0 ], [ 0, height/2 ], [ width/2,
height/2 ] ];
      break;
    // nobody joined; no positions
    case 0:
    default:
      break;
  }
  var namesElement = document.querySelector('#names');
  namesElement.innerHTML = '';
  for(var i = 0; i < names.length; i++) {
    var nameDiv = document.createElement('div');
    nameDiv.classList.add('name');
    nameDiv.style.left = positions[i][0] + 40 + 'px'; // added padding 40px
    nameDiv.style.top = positions[i][1] + 40 +
document.querySelector('#controls').clientHeight + 'px';  // added padding
40px plus height of controls
    nameDiv.appendChild(document.createTextNode(names[i] || 'An unnamed
driver'));
    namesElement.appendChild(nameDiv);
  }
  // update cars and cameras
  updateCars();
  // if client
  if(isHost !== undefined && !isHost) {
    // overwrite main render function with client one
    overwriteRender(socketId);
    // add .mobile class to controls to transform it
    document.querySelector('#controls').classList.add('mobile');
  }
});
```

This abstraction I made independently is the event listener for the **"**`updateUsers`**"** event in `public/js/game.js`. This function updates the client-side users array and the display every time there is a change to the array of users on the server-side (change in number of users or user names). It manages the positions of the names on the screen, calls the `updateCars()` function (from `/public/js/hostGraphics.js`) to update the array of 3D Car objects, and modifies the render function by calling `overwriteRender()` for smartphone controllers— a total of over 180 lines of code. The **"**`updateUsers`**"** event is sent out in three different instances by `server.js` (when a user joins and sets their name, when a user leave, and when a game is created). This manages complexity by grouping together many lines of code which only operate in tandem into a single instruction invoked with the **"**`updateUsers`**"** from the server. This abstraction reduces code redundancy, makes any future need to update the client-side users array(e.g., if the color of a user's car could be changed) very simple, and makes debugging the transfer of user data easy because the code is all in one place.