Technical Design Documentation

(By Robert Cale, Mathu Watts, Alex Baden-Hinsley and Anthony Lee)

Overview

Total development time 7.5 months from initial concept and publisher pitch to release build.
Developed in C# and the Microsoft XNA Framework due to familiarity with the group.
Distribution: Online download and physical disc.
Many game mechanics will be implemented with unfamiliar technology areas to the group so delays are possible.
Graphics are 3D models and sprite based.

Hardware

- Platform
  - Windows 7 64-bit, UWE Games Lab Arcade machine (running Windows 7)
- Supported resolutions
  - 1024x768 minimum required by Arcade machine
  - Custom resolution definable by user on Windows platform
  - Developed at 1280x720, 1024x768 and 1920x1080.
- Multiplayer
  - 4 players
- Sound systems required
  - Mono or stereo system
- Storage required
  - 50MB
- Hardware required for development
  - Monitor of minimum 1024x768 resolution.
  - Audio hardware installed.
  - Mouse and keyboard for writing code.
  - Keyboard and Xbox 360 gamepad for testing.
- Software required for development:
  - Visual Studio 2010
    - Required to develop with the Microsoft XNA Framework
  - Microsoft XNA Framework 4.0
    - Required to develop games built with the Microsoft XNA Framework
  - Image manipulation tool
    - Required to create the sprites and textures used in the art of the game
    - GIMP and Adobe Photoshop CS6 used during development
  - Sound editing tool
    - Required to convert audio files into a format support by the Microsoft XNA Framework
    - Audacity used during development
  - Text editor

        - 3D models files are created by hand by typing the model data directly to the text files
    - Windows 7 Operating System
        - Required to run and test the build for target platforms
    -

States Overview

- START
    - Load content files
    - Create instances of players
    - Moves into NEW_RACE
- MENU
    - Counts down to start of first race
    - Let players generate a new car
    - Let players veto and generate a new track
    - Displays the stats of each player's car
    - Starts the countdown music
    - Moves into MENU_TIMEOUT once count down is finished or skip buttons is pressed
- MENU_TIMEOUT
    - Place the first bomb on the track
    - Fades out music
    - Drops players onto the track
    - Moves into START_PLAYING
- START_PLAYING
    - Start the gameplay music
    - Moves into PLAYING
- PLAYING
    - Update particle effects
    - Finds a random player to set the flag to
        - If nobody already owns the flag
    - Updates the bomb
    - Find and respawn players who have died after a cooldown period
    - Check for collisions
    - Moves into PRE_VICTORY_LAP once a player has crossed the finish line
- RACE_OVER
    - Remove power ups spawned in the previous race
    - After the 3 races
        - Play the game over music
        - Move to OVERALL_WINNER
- NEW_RACE
    - Generates a new track
    - Sets players back to the starting line

    - Reset player status
    - Resets per race score
    - Resets the sound manager
    - Resets the cameras
- VICTORY_LAP
    - Displays the end of race scorecard
    - Moves to RACE_OVER after 5 seconds
- PRE_VICTORY_LAP
    - Store the player's race score to their total score
    - Moves to VICTORY_LAP
- OVERALL_WINNER
    - Shows the final scoreboard displaying the winner after the 3 races
    - Moves to NEW_RACE after the scorecard animation is finished
- PRE_MENU_TIMEOUT
    - Stop cars moving before race has started
    - Moves to MENU_TIMEOUT

Track Generation

The procedural track generation follows the algorithm:

1. Reset the track variables
2. Generate a new main curve for the track to follow
    a. Roll a random number
    b. Use this random number to determine the angle the next point to move off to
3. Generate the offset curves that the model will be built out of from the main curve
    a. Offset the main curve by the set widths and heights of the ramps and walls
4. Build the 3D model from the offset curves
    a. Add all the points of the curves to a vertex buffer
    b. Add the each sequential vertex to the index buffer
        i. A triangle list is used to allow this
    c. Set the colour of the vertices
5. Build the collision volumes
    a. Build a list of triangles from the vertices and indices
    b. Build a bounding box covering each sequential set of triangles

Car with track collision

The following algorithm describes how each the collision with the track is detected with each car and how they're handled after the fact.

1. Run through a broad phase first to reduce the number of collision tests
2. Check whether the car's whole bounding sphere is intersecting with any bounding box
3. For each intersection

    a. Loop through each triangle associated by that bounding box and check for an intersection between the triangle and the car's big bounding sphere
    b. Result the first triangle found to intersect with the car
4. Give this triangle to the car to handle the collision response
5. Cast a ray from the car's position to the triangle's plane to get the distance to the floor
    a. A plane is used as they are more reliable with intersection tests
    b. Cast a ray to the triangle instead if the first test fails
6. If the distance to the triangle is smaller than the car's smaller track bounding sphere's radius, then the car is in the ground
    a. If not in the ground, end function
    b. OnGround = false
7. In which case
8. Find the angle to the track by getting the dot product between the car's velocity and the triangle's normal.
9. Find the distance needed to push the car back so it's not clipping with the floor
    a. Radius of the bounding sphere - the distance to the triangle
    b. The vector to push back used is the triangle's normal
10. Push the car back
11. If the car is going into the track steep enough and the car's velocity Y is fast enough
    a. Bounce the car off the triangle's normal
    b. Reduce the velocity of the car to 75% * the car's Elasticity property
12. Else slide the car along the track
    a. Project the car's velocity into the triangle normal
    b. Redirect the velocity along the triangle
    c. Rotate the car's model up so it matches with the triangle's normal
        i. A cross product with the car's up vector and the surface normal will give us an axis to rotate along
        ii. The dot product between the up vector and the surface normal is the amount of angle to rotate

Car Handling

- The car stores an angle to face towards
- A forward direction vector is used to calculate which direction to direct the velocity
- Pressing the keys for each rocket turns them on
    - A force is used to calculate the new acceleration
        - The rockets' forces are culmulative
- Boost
    - If no rockets are turned on
        - Apply a force negative to the forward direction
    - If both side rockets are turned on
        - Apply a force to the forward direction
    - If only left rocket is on
        - Apply a force to right of the forward direction

        - If only the right rocket is on
            - Apply a force to the left of the forward direction
- Car collisions
    - Find the distance the cars are clipping through each other
        - Push them back
        - Normalize (Position - otherCar.Position) gets the direction
    - Add boost to each car's boost meter

Car Generation

This describes how each car model and stats are randomly generated

1. Each car part holds a list of available models
2. Randomly pick a model for each car part
3. Randomly pick a scale for each car part
4. Use the dimensions stored in each model file to offset each car part
5. The stats for the cars are based off the stats stored in each model file and scaled according to the scale of the car part.

Particles

The particle library Dynamic Particle Systems Framework was imported and used as a basis for the particle engine.