Marlena Skall
08/05/2020
Dice LA

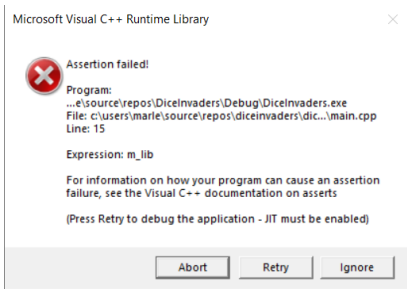
Design Doc: Dice Invaders

Setup / How To Run:

- New Project -> C++ Console Project (using Microsoft Visual Studio 2017)
- Transferred files from provided zip folder to new repo (files included: "data" folder, DiceInvaders.h, DiceInvaders.dll)
- How To Run:
  - Unzip folder, find .sln and open project
  - Make changes in Properties (if not already set):
    - `Linker -> System -> Subsystem -> Windows(/SUBSYSTEM:WINDOWS)`
    - `General -> CharacterSet -> Use Mutli-Byte Character Set`
  - Set Debug x86 or Win32 and run on Local Windows Debugger

Issues/Challenges In Initial Setup:

- (*) Using other options for subsystem like `Console(/SUBSYSTEM:CONSOLE)` threw errors
- Had to make changes (listed in Setup) to Properties for solution to properly run — initial example.cpp file threw errors (Window popup: Assertion Failed! Expression: m_lib)



- NOTE: If I incorrectly setup the project, I'd love to know how to do it properly

Project Design/Layout:

- Design centered around concepts from ITP380 class at USC (emphasis on Actors and their Components)
- Using the "Component Model":
  - Each game object "has-a" collection of components
  - i.e. Actor has vector of components (`mComponents`) with base functionalities
  - BUT, there exists the ability for derived classes to override some functions
    - ex. `OnUpdate(), OnProcessInput()`
- Actors:
  - For every object in the game, that you want to program behavior on, it will be an Actor
  - Each object is an Actor, but may have different behaviors/actions
    - ex: The player and the aliens are both actors, but the player can fire rockets and only moves left to right, while aliens drop bombs and descend down the screen
  - To help in the creation and deletion of Actors, each Actor has an `ActorState` enum where an Actor is either Alive, Paused, or set to Destroy (See Game Loop UpdateGame)
- Components:
  - Every Actor has a set of Components
  - In DiceInvaders, these components handle movement and the creation of the sprite/ appearance of the actor
  - Types of components used in this game are `MoveComponent` and `SpriteComponent` derived from `Component`

main.cpp:

- I created a main.cpp file to handle initialization of the library, with an instance of `Game`
- main checks that the library and instance of Game are properly initialized and then calls `Game::RunLoop()` to start handling the actually functionality of the game
- At the end `Game::Shutdown()` is called to unload data and `system` is destroyed

The Game Loop and Game Class:

- Created a Game class to handle all things "game" related (initialization, loop, shutdown)
- Using the provided library, I was able to create a main loop, modeled off of the example.cpp code, but designed in a way that would demonstrate clear game flow
- The main loop consisted of three main functions: `ProcessInput(), UpdateGame(), GenerateOutput()`
- These three functions are wrapped in a while loop calling `system->update()`
- Main data structures involved in this loop are the list of Actors and Sprites created
  - `ProcessInput()`
    - Where key presses are handled
    - From `system->getKeyStatus()`, can send keys to Actors and their components to process the key press
  - `UpdateGame()`
    - Where `float deltaTime` is calculated, which aids in use of timers and movement for the rest of the game, passed to `OnUpdate(float deltaTime)`
    - Manages all created actors stored in `mActors`, updates them and their components
    - Gathers any Actors that are set to `ActorState::Destroy` and deletes them
  - `GenerateOutput()`
    - Once all updates have been made (key presses processed, actors updated and dead actors removed), the game will actually display everything to the screen
    - This is where the `mSprites` vector comes in
    - Each actor has a SpriteComponent, we call `Draw()` on it to draw to screen (where Draw() uses the `sprite->draw()` function

Movement:

- Combination of "frame-locked" movement and movement based on `deltaTime`
- The player, bombs, and rockets, relied on their MoveComponent to update position by a specified speed * deltaTime
- The aliens moved by a specified number of units each cycle (based on a timer that reset every time it ran down to 0)

The Player:

- Player class relied on MoveComponent and key presses to move from one side of the screen to the other
- Used leading edge detection method to handle firing of rockets as well as a short timer between rockets launched to make it slightly more challenging when facing the aliens
- Player can also be upgraded
- (See "Additional Features")

Creating and Managing the Aliens:

- Loading the enemies from .txt file
  - To support creativity and allow the designers flexibility, I thought about instantiating a level by reading from .txt a file
  - This way, aliens were organized into different patterns so that a new/different level appeared as the player advanced through the game

- Once all the level.txt files have been run through (increasing by 1 on each call to `LoadNextLevel()`), levels are randomly selected among the files provided
- (See "Make Changes" below on creating your own level)

- To organize creation of the aliens I had an `Enemy` class, `MysteryEnemy` class, and `EnemyBlock` class
  - `Enemy Class`
    - Specified green or red enemy in constructor
    - Handled creation of bombs to be dropped
    - In OnUpdate(), checked for collisions with the player
  - `MysteryEnemy Class`
    - Inherits from Enemy class
    - From seeing the 1978 Space Invaders clip, I wanted to add the mystery alien that flies across the top of the screen
    - Rather than move like other enemies (left to right while shifting down the screen), the MysteryEnemy would only travel left and right at the top of the screen
    - OnUpdate(), used the MoveComponent to set speed modified by deltaTime that changed direction every time the enemy hit the sides of the screen
    - Relied on `mDir` to change directions
    - Appears randomly during a level based on a timer
  - `EnemyBlock Class`
    - Created at the end of loading the level (read from level.txt file)
    - OnUpdate, used the vector `mEnemies`, which contained all enemies created in the game, to check collisions with the sides of the screen
    - If an enemy were to move beyond the boundary, the EnemyBlock would shift all enemies down by a specified amount (in this case 32.0f)
    - If not, then the enemies would shift to the left or right (based on mDir)
    - The block moved based on `mTimer` which gradually got shorter as the levels progressed, but having a timer gave off the appearance of the EnemyBlock inching its way to the player
    - `mBombTimer` and `RandomBombDrop()` control the actions of the aliens in the block, where at random, an alien will drop a bomb

Handling Collisions:
- Used a distance check to determine whether an Actor collided with another Actor
- This version calculated the distance between two actors by extracting the x and y components using `GetHorizontal()` and `GetVertical()`
- Enemy class handled collisions with player
  - Since the EnemyBlock data is updated based on the `mEnemies` vector, I decided that when EnemyBlock instructs all enemies to shift left/right/down, that each enemy would do its own check against the player
- `Bomb` class handled collisions with player
- `Rocket` class handled collisions with enemies

HUD (Player Feedback):
- I wanted to take advantage of the `drawText` function provided so at the top of the screen, the player can see their current score, the current high score, the level they are on and how many lives they have
- Wanted to provide some player feedback to draw attention to different parts of the screen (even for a short moment)
  - When the player loses a life, they lose ship icon representing an extra life
  - When the player destroys a MysteryEnemy, the bottom right of the screen says "Player Upgraded" and disappears if the player is hit

Additional Features:

- MysteryEnemy
- Player Upgrade
  - Shooting MysteryEnemy allows player to fire two rockets
  - If player is hit with upgrade on, the player doesn't lose a life but loses the upgrade
- Every 1000 points earned, gain back a life if one was lost
- If there is one remaining enemy in the EnemyBlock, the enemy's speed in increased and it makes its way down the screen faster
- Loading and saving high score to file
  - On GameOver, if the player's score is higher than the high score, that score is saved as the new high score to highscore.txt
  - The next time the player starts the game, that score will be read from highscore.txt and loaded as the current high score to beat
- Rockets can destroy bombs (See "Make Changes")
- Customizable Levels (See "Make Changes")
- Created two log files to output system information and game information while programming this assignment (added `#define DEBUG` to DiceInvaders.h) (See "Make Changes")

Make Changes

- Create your own level (located in data/Levels folder)
  - To create a new level, the format follows capital letters for red and green aliens ('R' and 'G' respectively) and '.' to indicate spaces between enemies or between rows
  - The .txt format is "data/Levels/level000.txt" where "000" supports potentially hundreds of levels but to load them correctly would mean following after the last level created
    - ex. next level is "`data/Levels/level005.txt`" and `LEVELSPROVIDED` in Game.h needs to be set accordingly (i.e. from 000 to 005, that would be 6 levels provided)
  - LoadLevel() should look for errors in file creation (i.e. incorrect rows/columns)
- Change the starting level:
  - go to `Game::Initialize()` and change line 38
  - set `mCurrLevel` to any .txt file in Levels folder (ignore test files 0001 and 0002)
- Have rockets destroy bombs: go to `Rocket::OnUpdate()` and uncomment lines 29-37
- Clear the high score
  - highscore.txt located in data folder
  - To reset the score you can clear the contents of the file or to load in a score you can write in the file
- See Log info: in DiceInvaders.h, change line 4 so that the DEBUG flag is set to 1

Future Thoughts / Reflection:

- If I had more time:
  - Add fun animations to enemies and explosions for player feedback
  - Create a header file for vector operations
  - Had different enemies drop varying amount of bombs (ex. red enemy fires 2 bombs)
  - With much more time I would have liked to have simulated variable trajectories for enemies (similar to Galaga from Namco https://www.youtube.com/watch?v=WSF0z6mq46A — see minute 0:30)
- Thoughts on how I would improve the current code structure
  - Potentially would have modified/added a new class to handle GameData specifically so that the Game class could be primarily focused on generating Actors and their components and initializing the system
    - I had the Game class handle a lot of the data (score, player lives, player upgrade) because every time a new level is loaded, all the actors are deleted