Graphics I Report: 100053181

Contents

Game Overview

Colour Up! is a 2D platformer game with a twist. The game preserves the basic platformer mechanics such as moving and jumping from one platform to another and the goal is to reach the end which is high up from the starting point. The game also utilizes a unique game mechanic, though, which I call 'colour blending'. The main character – *C-MIKE* - has the ability to change the background colour of a level from the default black to one of the other subtractive colours of the CMY colour space: that is cyan, magenta or yellow. Most of the platforms are solid white but some of them are cyan, magenta or yellow (*The Scan-Lines)*. By changing the background colour these platforms 'blend' and as a result are no longer platforms but just part of the background. Because of that, the player has the ability to pass through the platforms and make progress. The game rewards the player for the time they take to complete the level, amount of collectibles they pick (*The Lambdas*) and at all times keeps track of the current altitude of the player. Player dies when they interact with *The Alpha* - corrupt force of transparency in two forms. The first is a big alpha cloud constantly moving up and forcing the player to do so, too. The other are *The Alpha Minions* – enemies found on some of the platforms. The player cannot defeat them, only avoid them. *The Mixes* are power-ups that temporarily increase the maximum jump height.

'Colour Blending'

Preparation

Since the game is all about the colours I had to create a tool to quickly switch between the colours without having to use pure RGB values. Colour.h contains an enum Color (i.e. CYAN, MAGENTA) and a corresponding parallel array of GLfloats holding the actual RGB values that OpenGL recognizes. The file also contains a number of functions streamlining the 'colour blending' process.

Changing the BG colour

By pressing **[C]**yan, **[M]**agenta or **[Y]**ellow on the keyboard, the current background colour gets changed to the desirable value. At the same time the timer starts and counts to 2 seconds. This is the duration of the background colour change. During that time a bool flag prevents the player from tricking the system and also from constantly pressing the button to prolong that duration to infinity.

Update blending

Each object in the game has a member variable specifying its colour. At the beginning of every *update()* call, the function *checkBlending()* traverses the vector of ***CollidableObjects*** (any object that has a bounding box and is not a player) and compares the current background colour to the one of the object. If they match, the object gets marked as 'blended' and is removed from the collision detection check. The moment the colour of the background becomes black again, the said object becomes collidable again.

ActivityStates

The game utilizes ***ActivityStates***. ***Activity*** is a virtual class and three classes inherit from it – ***StartGame***, ***PlayGame*** and ***EndGame***. ***ActivityManager*** is a class that is responsible for handing switching between those states. All of these states share the four important functions that get called at every frame.

init() – initializes all the variables and sets up the state (only called once)

update() – updates the variables

updateInput() – processes the keyboard input

draw() – draws the scene

At the beginning of the program those three states pushed into the vector of states in the ***ActivityManager*** that gets initialized from the main.cpp. In the main.cpp there are also

init()/update()/draw() functions that always call the corresponding functions on whichever state is at that given frame set to be the active one. By default, the first one to be initialized is the ***StartGame*** state which is a start menu that also includes a user manual. By pressing the spacebar the flag is set off. During the ***ActivityManager***'s *update()* function this change is picked up and the next state is loaded. ***PlayState*** is the main game state. When the player dies/finishes the level, the ***EndState*** is loaded. It shows the high-score, time of completion and enables the player to start again.

Collision Detection and Response.

The collision detection that I used in this game is a simple AABB algorithm because of the nature of the game which deals only with axis-oriented squares and rectangles.

Initially I settled for the brute-force ('hacky') collision response mechanism in which I would simply stop the player movement and disable gravity when the collision happened but I quickly realized that if I wanted to make a functioning game with robust features I could not take any shortcuts. I redesigned the whole process. My collision detection/response system predicts where the player is going to be and prevents the collision from happening. This is the order in which the functions are executed in the *update():*

1. *updateBlending()* – mark objects as 'blended' (not collidable)
2. *updateInput()* – get the keyboard input and move requests
3. move the obstacles
4. calculate the requested speed vector for the player based on the move requests
5. *checkForCollision()*
6. move the player by the modified speed vector

pseudo-code for checkForCollision()

```
create a temporary bounding box (a clone of the player's one)

loop through all the obstacles (excluding the ones that are 'blended')

      translate the bounding box by the requested speed vector

            if (there is collision and the obstacle is not 'blended')

                  get the minimal distance to push the player out of the
                  object (in four directions) and add a small value to it
                  so that they do not collide anymore

                  find which one of those distances is the smallest, divide
                  it by delta-time to get the modified speed value and
                  add/subtract it (based on the direction) to/from the
                  requested speed vector. If the player has to be pushed up
                  (they hit the floor) set the flag 'jumping' to false.

                  switch(obstacle type)

                        perform additional tasks i.e. killing the player if
                        the obstacle is of type ENEMY or DEADLYPLATFORM
```

Textures

All the textures used in the game were created by me.

Game Manual

Controls:

Left arrow – move left

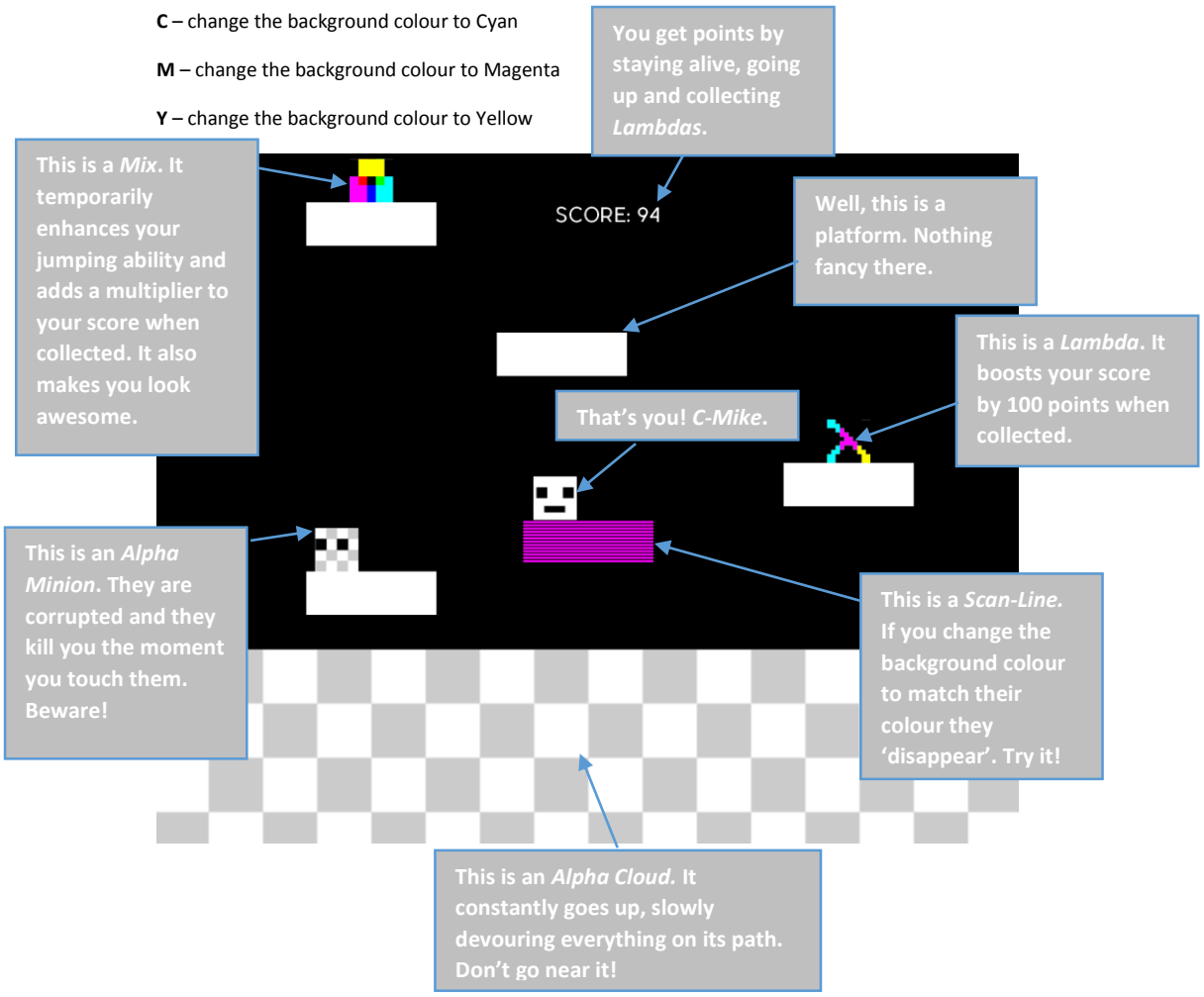
Right arrow – move right

Up arrow – jump

C – change the background colour to Cyan

M – change the background colour to Magenta

Y – change the background colour to Yellow



Objective:

Go up, collect *Lambdas* and don't get killed by *Alpha Minions* and *Alpha Cloud*.

Tips:

Use *C-Mike*'s super power to temporarily change the background colour to pass through *Scan-Lines.*