Assignment Two

PROG2215

March 21, 2018

Wesley Martin 7529183

TABLE OF CONTENTS

INTRODUCTION

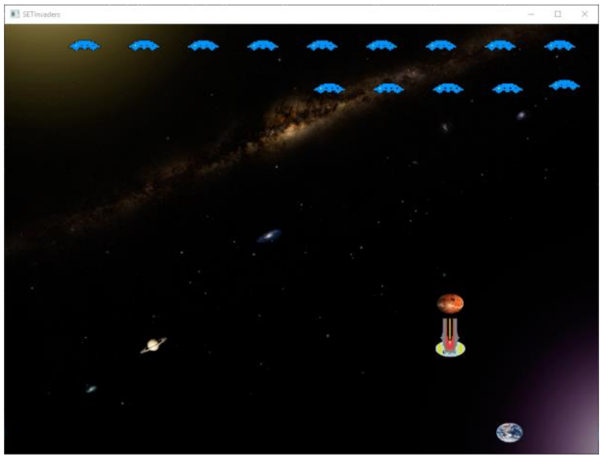
This document is a description and a summary of my attempted implementation of the SET Trek II game. This document is split up into two large sections. The first section is a summary of each functional requirement of the program, along with screen shots demonstrating those requirements in action. There is a one to one correlation between the sections describing the functional requirements, and the sections in the second half of the report which contain code from the assignment that demonstrate how each requirement was implemented.

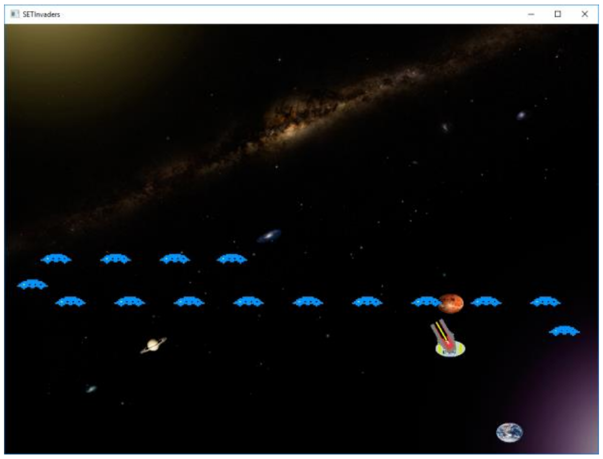
FUNCTIONAL REQUIREMENTS ANALYSIS

REQUIREMENT 2.1: ENEMY MOVEMENT AND ANIMATION

REQUIREMENT 2.1.1

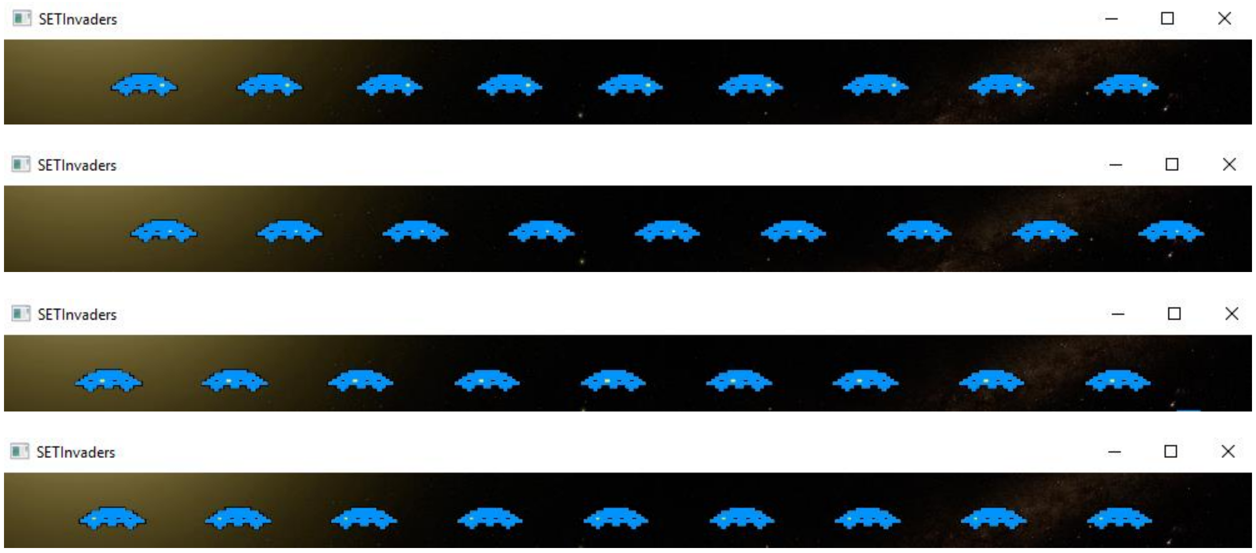
To animate the ships I am use a two stage check. The first one involves looking at their current horizontal location and comparing it to the game grid. If they are within one of the two outer grids, then the program starts looking at their vertical location. It captures their current vertical location, calculates the vertical movement based on the speed, and then checks to see if their current row is different from their previous row. If it is, that means that it can begin moving horizontally again. The ships horizontal and vertical speeds are calculated based on the width of the columns and the heights of the rows. The location of the ships is recalculated every 50th of a second.





REQUIREMENT 2.1.2

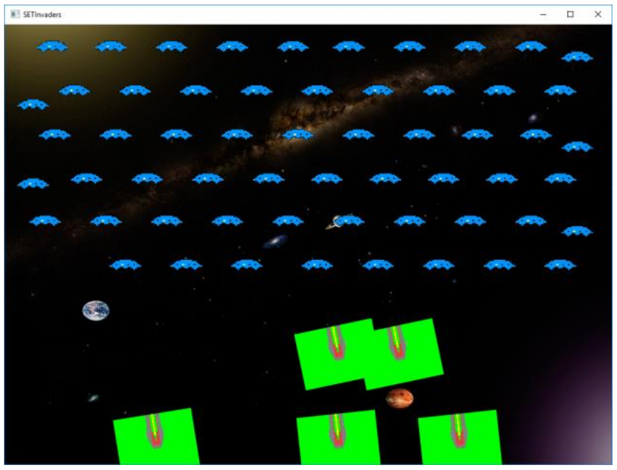
To display the animation of the ships, I added a vector of bitmap images as a member of each ship object. That vector contains all of the possible images, in left to right order, that are needed to render the ship. The images are then selected by moving to the next one in the vector every 25 ticks, or every half a second, as there are 50 ticks per second. This aligns with the ships movement of moving to a new tick every half a second.

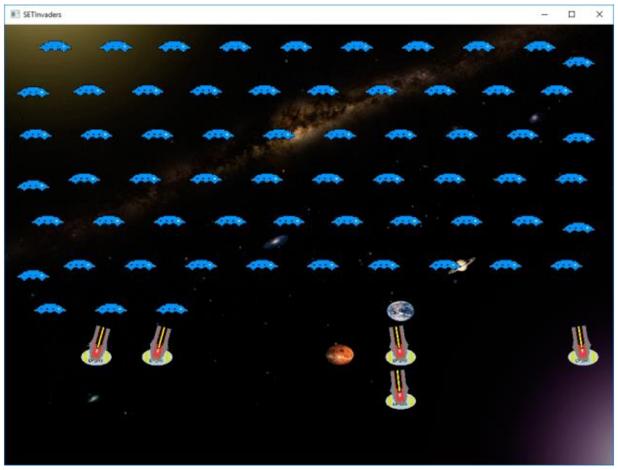


REQUIREMENT 2.2: DEFENDER SATELLITE/DSAT

REQUIREMENT 2.2.1

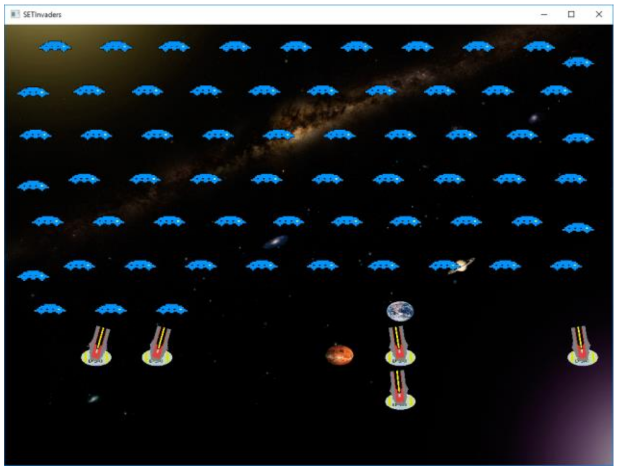
To render the defenders, I modified to the program slightly so that every program is made up of a static base image and a rotating turret image. Both images are drawn using the same green screening process as the previous assignment, which involved stepping through each pixel in the image and making any pixel that is pure green invisible. The code for this process can be found again in the documentation. The defender inherits from the same parent as the enemy spaceships, and is drawn to the screen in the same way.

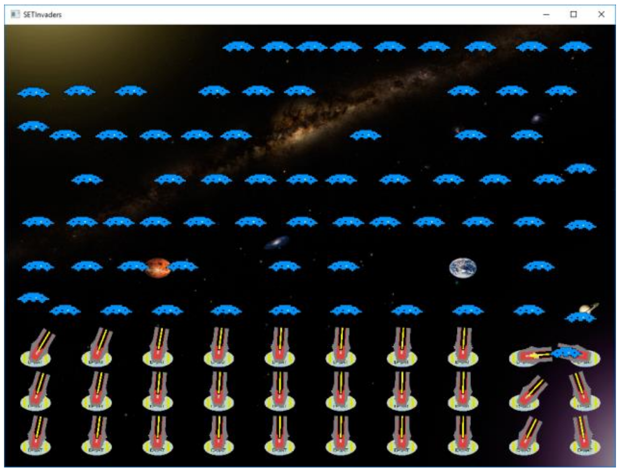




REQUIREMENT 2.2.2

The process for randomly placing defenders is fairly simple, due to the nature of the GameGrid class. Every time I want to place a new obstacle, I simply ask the GameGrid for a new empty slot, give those x and y values to the obstacle object, and then mark that slot as full. The code for this can be found in the reference code, under “Randomize Defender Placement”. The grid itself does not have knowledge of the size of the render area, rather it just stores the index location, so drawing the turrets at the proper scale involves grabbing the render width and height. Those numbers are given to the draw function, which then runs its formula to calculate the x and y origin, and the width and height. The calculations for this can be found in the example code.





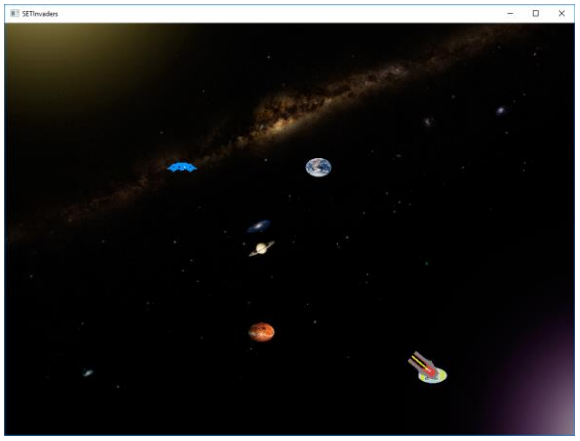
REQUIREMENT 2.2.3

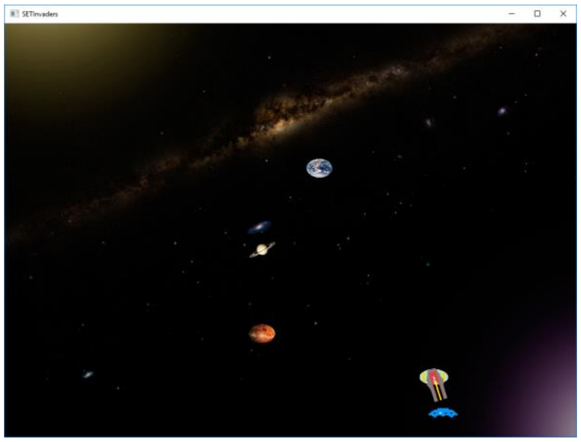
DISTANCE

To calculate the distance to the nearest enemy, I am creating a vector based off of the locations of the defender and the enemy in question. I pass the enemy as a reference to the calculate distance method, which in turn is doing some fun vector math. The method subtracts the x and y value of the ship from the x and y value of the defender, giving a new point that is, relative to the plane origin, the same as the vector between the two game objects. From that point, I am using the XMVector2Length method to get the length of the vector, and then the XMStoreFloat method to pull the distance as a float out of the vector object.

ROTATION

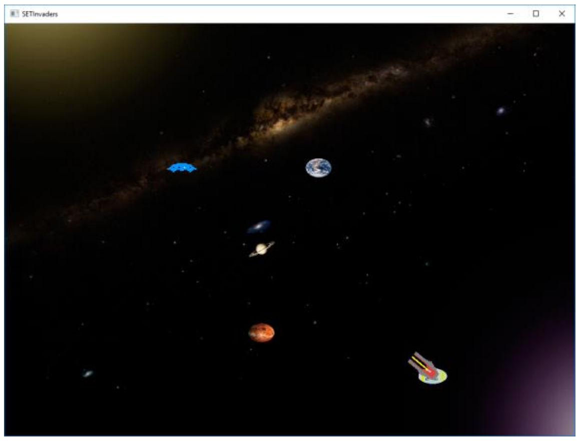
I am calculating the angle in a similar fashion to the distance. The first step is two create two vectors, a normalized one starting at the vertex, and a vector representing the difference between the defender and the target. I then normalize both vectors, calculate the dot product of the vectors, calculate the arc cos of the dot product result, and then convert that into degrees.

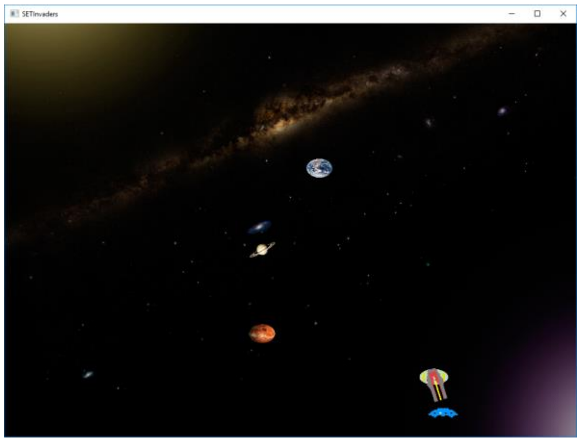




REQUIREMENT 2.2.4

When the method is called that is used to calculate the rotation matrix, I first calculate the distance to the enemy spaceship and then compare that to four times the height of a single row of the game map. If the distance is less than the sight of the enemy, I then I follow a set of steps very similar to requirement 2.2.3 – Rotation. Once I have calculated the angle between the two objects, I simply create a new 3x2 matrix based off of the calculated angle and the midpoint of the spaceship. This will allow the ship to rotate around the midpoint of the ship and angle equal to the calculated angle. The angle will continue to be updated and point to the ship for as long as the ship is below the highest row, and still on the screen.





REQUIREMENT 2.3: DEMO

For the purposes of the demo, I will run the program in the simplest way possible, with one enemy, one defender, three obstacles, and no other complications.

CODE DOCUMENTATION

REQUIREMENT 2.1: ENEMY MOVEMENT AND ANIMATON

REQUIREMENT 2.1.1

GAME TIME TRACKING

```
/**
* \brief Gets the number of milliseconds that have passed
* \return The number of milliseconds that have passed
*/
int GameController::GetGameTicks()
{
#define kMILLISECONDS 1000
	timeb tb;
	ftime(&tb);
	int nCount = tb.millitm + (tb.time & 0xfffff) * kMILLISECONDS;
	return (int)nCount / kMILLIS_PER_TICK;

}
```

ENEMY MOVEMENT PLANNING

```
bool Enemy::Move(LevelGrid* grid, float gameWidth, float gameHeight)
{
	int currentRow = kDEFAULT;
	bool ret = true;

	// Check to see if spaceship is at either edge of screen
	if (this->GetX() <= kLEFT_EDGE || grid->ColumnFromX((int)this->GetX(), (int)gameWidth) == grid->getRows() - 1)
	{
		// Move ship vertically
		currentRow = grid->RowFromY((int)this->GetY(), (int)gameHeight);
		this->SetY(this->GetY() + this->GetYSpeed());

		// Check to see if spaceship has fully moved down to the next row
		if(grid->RowFromY((int)this->GetY(), (int)gameHeight) != currentRow)
		{
			this->SetXSpeed(this->GetXSpeed() * kINVERTED);
			this->SetX(this->GetX() + this->GetXSpeed());
		}
	}
	else
	{
		// Otherwise, move horizontally
		this->SetX(this->GetX() + this->GetXSpeed());
	}


	return ret;
}
```

REQUIREMENT 2.1.2

IMAGE SELECTION

```
/**
 * \brief Used to change the image of the spaceship
 * \param imageIndex
 * \return Indicates success of image change
 */
bool Enemy::ChangeImage(int imageIndex)
{
		bool ret = true;
		if(imageIndex < images.size())
		{
				this->SetImage(images[imageIndex]);
		}
		else
		{
				ret = false;
		}

		return ret;

}
```

CREATING A NEW ENEMY WITH GRAPHICS ARRAY

```
enemy = new Enemy();
enemy->SetYSpeed((float)this->renderHeight / (float)this->rows / (float)kENEMY_SPEED_SCALE);
enemy->SetXSpeed((float)this->renderWidth / (float)this->columns / (float)kENEMY_SPEED_SCALE);

// Setup the enemy images
char enemyString[kPLANET_FILE_STRING];
for (int i = kDEFAULT; i < kNUMBER_OF_ENEMY_FILES; i++)
{
		sprintf(enemyString, kENEMY_FILE_NAME, i % kNUMBER_OF_ENEMY_FILES +
				kFIX_OFF_BY_ONE_ERROR);
		enemy->AddImage(new BitmapImage(enemyString, this->gfx, false));
}
enemy->SetPosition(kDEFAULT + kFIX_OFF_BY_ONE_ERROR, kDEFAULT);

// Add the enemy to the list of sprites

sprites.push_back(enemy);
```

CHOOSE IMAGE BASED ON TICK COUNT

```
((Enemy*)sprites[i])->ChangeImage((this->ticks / (int)kENEMY_SPEED_SCALE) % kNUMBER_OF_ENEMY_FILES);
```

REQUIREMENT 2.2: PLACE PLANETS

REQUIREMENT 2.2.1

DEFENDER CREATION

```
// Set up the defender
for (int i = 0; i < kNUMBER_OF_DEFENDERS; i++)
{
		// The true as the third parameter implies chroma keying
		defender = new Defender(new BitmapImage(kDEFENDER_FILE_NAME, this->gfx, true));
		defender->SetTurretImage(new BitmapImage(kTURRET_FILE_NAME, this->gfx, true));
		sprites.push_back(defender);

}
```

APPLY CHROME KEY

```
void BitmapImage::GreenScreenFilter(BYTE* pv, int width, int height)
{
		for (int i = kDEFAULT; i < width * height * kSIZE_OF_INT; i += kSIZE_OF_INT)
		{
				if (pv[i + kRED_INT_INDEX] < kRED_THRESHOLD
						&& pv[i + kGREEN_INT_INDEX] > kGREEN_THRESHOLD
						&& pv[i + kBLUE_INT_INDEX] < kBLUE_THRESHOLD)
				{
						pv[i + kGREEN_INT_INDEX] = kBLOCK_IS_EMPTY;
						pv[i + kALPHA_INT_INDEX] = kBLOCK_IS_EMPTY;
				}
		}

}
```

REQUIREMENT 2.2.2

RANDOMIZE DEFENDER PLACEMENT

```
// Pick a new column for the defender
gameGrid->getEmptySlot(&x, &y, kHIGHEST_DEFENDER_ROW);

((Defender*)sprites[i])->GetTurret()->SetPosition(
		gameGrid->XFromColumn((int)x, (int)renderWidth),
		gameGrid->YFromRow((int)y, (int)renderHeight)
);

// Update the location of the defender
sprites[i]->SetPosition(
		gameGrid->XFromColumn((int)x, (int)renderWidth),
		gameGrid->YFromRow((int)y, (int)renderHeight)
);


gameGrid->fillSlot(x, y);
```

REQUIREMENT 2.2.3

DISTANCE TO ENEMY

```
/**
* \brief Used to calculate the distance between the defender and the give
* enemy spaceship
*
* \param enemy The enemy that the defender is targeting
*/
float Defender::CalculateDistanceToEnemy(GameObject* enemy)
{
		DirectX::XMVECTOR target = DirectX::XMVectorSet(enemy->GetX() - GetX(), enemy->GetY() - GetY(),
kDEFAULT, kDEFAULT);
		DirectX::XMVECTOR length = DirectX::XMVector2Length(target);

		float distance = kDEFAULT;
		DirectX::XMStoreFloat(&distance, length);
		return distance;
}
```

ANGLE TOWARDS ENEMY

```
// Get the vector to the target and the source
DirectX::XMVECTOR target = DirectX::XMVectorSet(enemy->GetX() - GetX(), enemy->GetY() - GetY(), kDEFAULT,
kDEFAULT);
DirectX::XMVECTOR source = DirectX::XMVectorSet(kDEFAULT, kINVERTED, kDEFAULT, kDEFAULT);


// Prepare normalized vectors
DirectX::XMVECTOR target_n = DirectX::XMVector2Normalize(target);
DirectX::XMVECTOR source_n = DirectX::XMVector2Normalize(source);

// Check angle
float targetAngle =
		DirectX::XMConvertToDegrees(DirectX::XMVectorGetY((DirectX::XMVectorACos(DirectX::XMVector2Dot(target
		_n, source_n)))));
if (enemy->GetX() <= GetX())
{
		targetAngle *= kINVERTED;

}
```

REQUIREMENT 2.2.4

MATRIX TRANSFORM

```
// Only do anything if the enemy is across the target row
if (CalculateDistanceToEnemy(enemy) <= sight)
{
	// Get the vector to the target and the source
	DirectX::XMVECTOR target = DirectX::XMVectorSet(enemy->GetX() - GetX(), enemy->GetY() - GetY(),
		kDEFAULT, kDEFAULT);
	DirectX::XMVECTOR source = DirectX::XMVectorSet(kDEFAULT, kINVERTED, kDEFAULT, kDEFAULT);


	// Prepare normalized vectors
	DirectX::XMVECTOR target_n = DirectX::XMVector2Normalize(target);
	DirectX::XMVECTOR source_n = DirectX::XMVector2Normalize(source);

	// Check angle
	float targetAngle =
		DirectX::XMConvertToDegrees(DirectX::XMVectorGetY((DirectX::XMVectorACos(DirectX::XMVector2
		Dot(target_n, source_n)))));
	if (enemy->GetX() <= GetX())
	{
		targetAngle *= kINVERTED;
	}

	// Calculate the rotation matrix
	D2D1::Matrix3x2F R;
	if (enemy->GetType() == ObjectType::Enemy)
	{
		R = D2D1::Matrix3x2F::Rotation(targetAngle, GetMidpoint());
	}

	// Apply the rotation
	turret->SetRotation(R);
}
else
{
	// Apply the default identity matrix
	turret->SetRotation(D2D1::Matrix3x2F::Identity());

}
```