

Arkanoid Test for Plunge Interactive

Octavio Martínez García (o.martinez.gar@gmail.com)

October 15, 2014

Overview

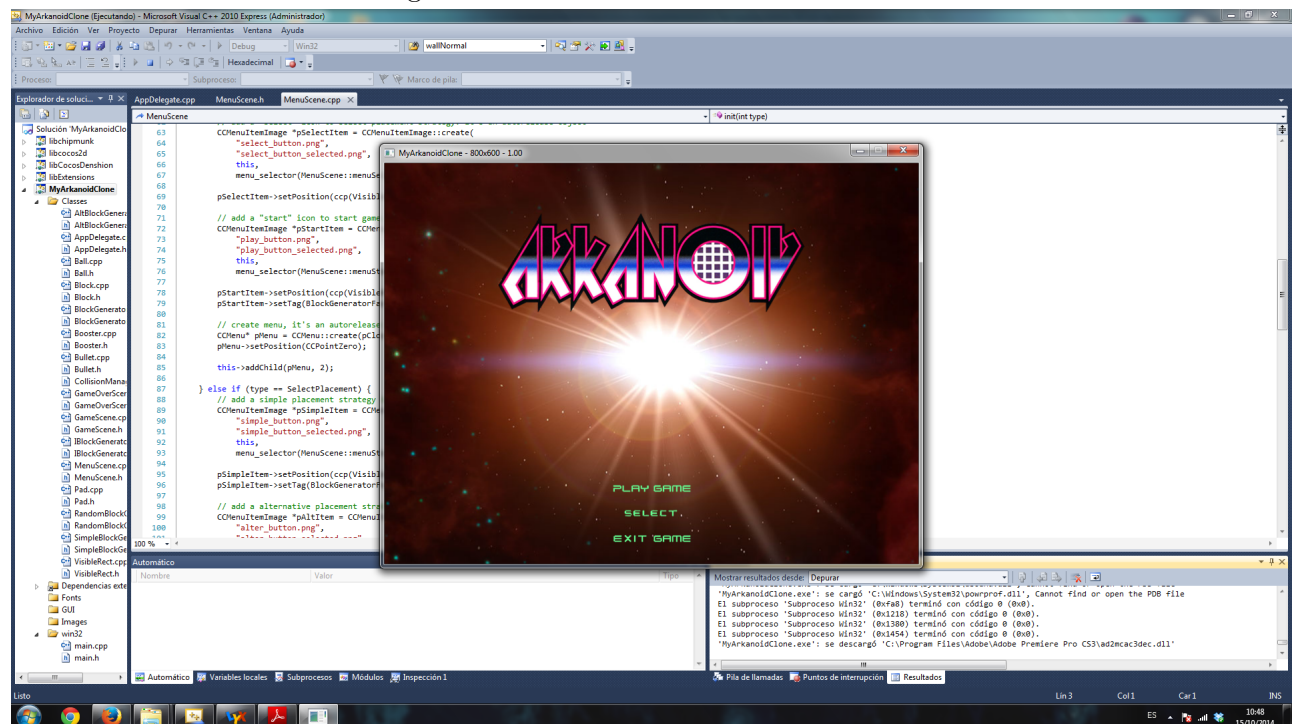
- The test required building an Arkanoid/Breakout clone from scratch using cocos2d-x v.2x and C++. Also, for the physics integration, we have used the Chipmunk library, as one of the game requirements was to forbid the use of Box2D. The rest features and mechanics are defined in the requirements pdf, and will be disclosed in the following section as well, as we describe the work done.

Review

The game is divided in 4 scenes which compose the game flow. Those scenes are described below:

Main Menu scene:

Figure 0.1: Main Menu Scene

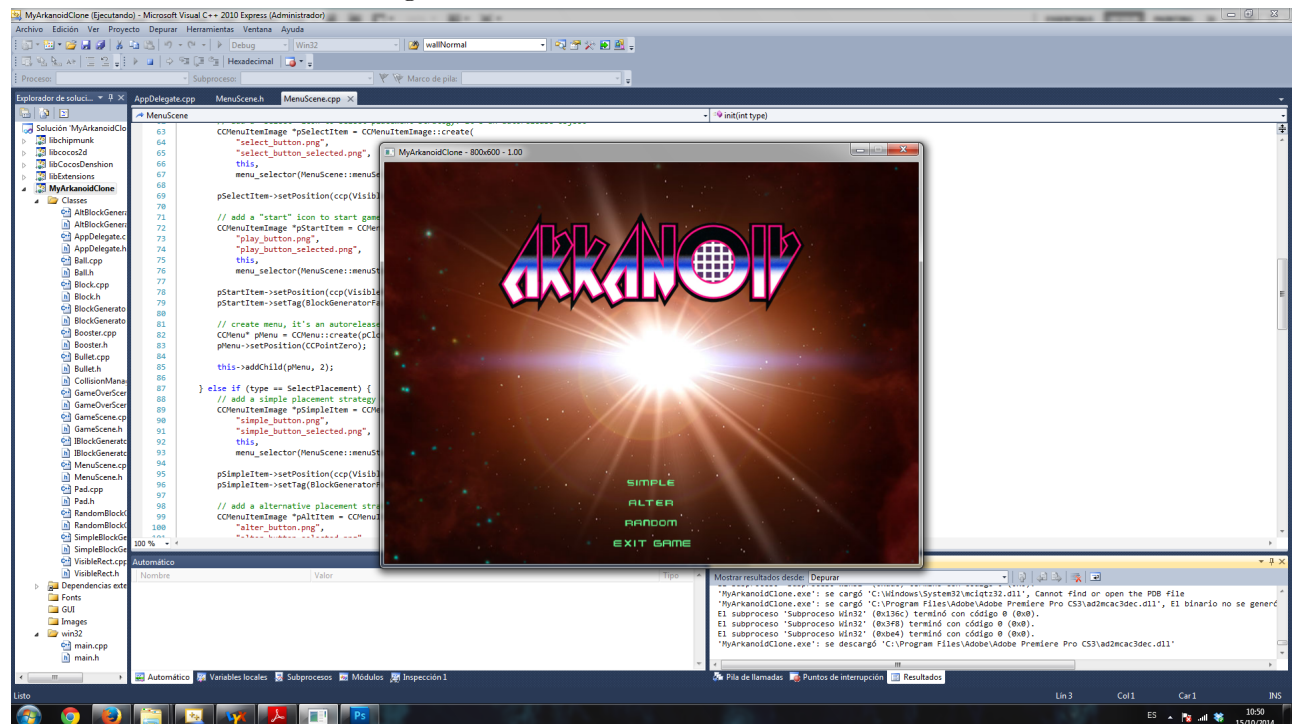


- it is the initial scene, which is loaded from the AppDelegate class and passed to the CCDirector to run it as the game is launched.
- it is encapsulated in the MenuScene class, being its purpose to present the game and to make the game menu available for starting the game.

- as it is shown it has 3 buttons, which have some menu callback methods attached providing the following functionalities:
 - Play game: starts the game with a Random block placement algorithm (default option).
 - Select: changes the scene to the Select Menu, which lets the player choose from 3 different block placement algorithms from menu items.
 - Exit Game: pretty self-explanatory, it quits the game.

Select Menu scene:

Figure 0.2: Select Menu Scene

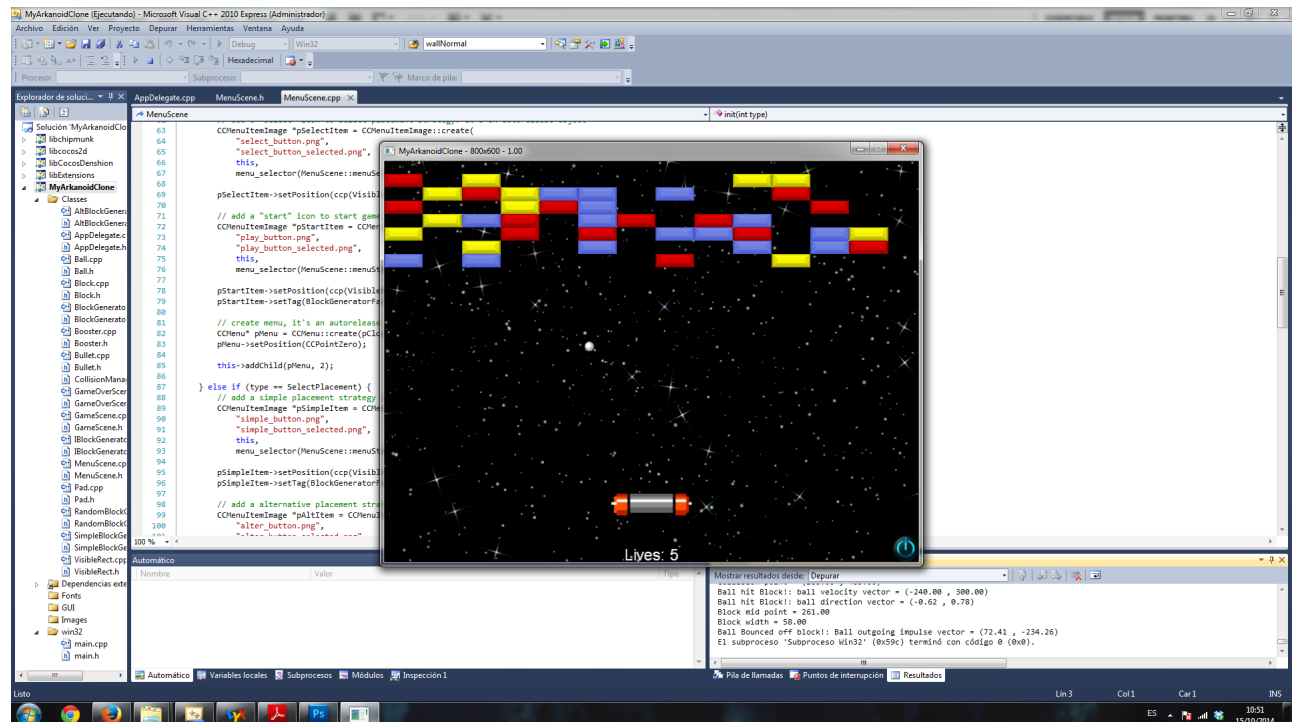


- if it is selected, by pushing the select button in the Main Menu, it presents a number of option buttons that let the player choose different block placement algorithms implemented, that lead to different block generation patterns.
- it is also encapsulated in the `MenuScene` class, and chosen by the `'type'` parameter passed on creation, providing this way a custom menu creation, and reusing the same class for various similar purposes (main menu or select menu).
- the 3 block placement strategies are as follows:

- Simple: it fills 3 rows with red (1 hp) blocks
 - Alter: it places 1-7 rows of random type blocks (blue(3hp), yellow(2hp) or red(1hp), placing alternatively 1 block, 1 hole, 1 block, etc.
 - Random: it places 1-7 rows of random type blocks, placing randomly a block or a hole.
- when the user click on some of those strategy buttons, a menu callback function is called, and the menu item tag is passed to the GameScene create method as a parameter, which indicates the block placement strategy that it should use.
 - as included in the requirements, the block placement strategies, are implemented following a Strategy Design Pattern. Thus, we have a base class (an interface), *IBlockGenerator* class, that has a pure virtual function '*generateBlocks*', implemented by several classes that comprise the actual placement algorithms, that are chosen at runtime from the select menu.
 - for creating the actual placement strategy, we use the '*BlockGenerationFactory*' class in the init method of the GameScene class, that has an '*IBlockGenerator* __generator*' attribute, and a static method '*generateBlocks*' that is called with a parameter that creates the actual generator strategy, calling its '*generateBlocks*' method.

Game scene:

Figure 0.3: Game Scene



- it is encapsulated in the `GameScene` class, which also has a *'type'* parameter passed on creation that lets the user choose from the different placement algorithms available. This class encloses all the core game logic, handling physics collisions, game loop (update function), touch events, etc.
- here, in the main scene, is where the actual game takes place. It's composed of the typical arkanoid elements. All these game elements are classes derived from the `CCPhysicsSprite` class, and all have a similar static creation method, which marks them as autorelease objects, and calls their inner `init` method, initialising several attributes, and encapsulates the chipmunk physics initialization, creating shapes and bodies attached to them, adding them to the `cpSpace` (chipmunk physics space), and doing several additional custom initializations.
- seeing this we have thought to refactor that initialisation into a base class and make them inherit from it, but due to the particular physics init differences (mass, moment of inertia, elasticity, tags, velocity, etc.) that they have, we have opted to leave them as they are.

– Pad: 

- * a sprite with physics logic attached with which the player should prevent the ball from falling making it bounce against the blocks, to break them and win the game consequently.
- * it can be moved horizontally by dragging it sideways with the mouse cursor or with touch if able (touch / click event - '*ccTouchMoved*'). Inside the touch handler function, we get the next position, by getting the touch point, and if it's a valid position (i.e. pad is inside screen) we update the pad position
- * also, if we have caught a bullet booster, the pad will transform into the laser pad, and we will be able to shoot bullets upwards to destroy blocks for a 3 sec lapse. In this form, we can tap on the pad sprite to shoot, which is managed by the '*ccTouchBegan*' handler, that internally calls the '*shoot*' function on the gameScene `_pad` attribute, that instantiates bullet objects on the position of the pad and give them an upwards physics impulse.

– Ball: 

- * an ever bouncing ball which damage and destroy blocks on collision, and bounce against every other game shape. If the ball falls from the bottom of the screen the player loses 1 life.
- * the ball has different behavior depending on the shape with which it collides, that is encapsulated in the 'bounce' method, called from the collision handlers registered in the GameScene, which calls different ball's private methods to handle different situations.
- * if it collides with the game walls (invisible static segment shapes that delimit the game bounds) it bounces back with a 90° angle direction, unless it is the bottom wall, in which case, it is destroyed, the player loses 1 life (only if there's no extra ball on screen), and a new ball is created from the middle of the screen with a random direction impulse (but upwards).
- * else if it collides with a block or the pad, it bounces back in a angle that depends on the proximity of the impact point to the mid point of the shape, using this formula:
- *

$$xDirection = \frac{impactPt.x - shapeMidPt.x}{shape.width/2}$$


$$yDirection = -yDirection$$

– Blocks: 

- * these are the shapes that we have to destroy to win the game, which are placed by the block placement algorithm selected. They have different types with different hit points associated (blue(3hp), yellow(2hp), red(1hp)).
- * when the ball collides with them they receive 1 point of damage, changing its color and lowering its hp, and destroying it (dealloc its body and shape from chipmunk space, and removing it from its parent, and thus freeing it, as its an autorelease object) if its life reaches 0. All this is done in the *'takeDamage'* method, called in a post physics Step collision callback registered in the GameSpace.
- * also if its destroyed, it can release a random booster type randomly (with a 25% chance), done in the *'releaseBooster'* method, instantiating a booster physics sprite, with an initial downwards impulse (as we have set the game to have 0-gravity, because we don't make use of it, except in this particular case, and we can simulate it with a physics impulse force)

– Boosters: 

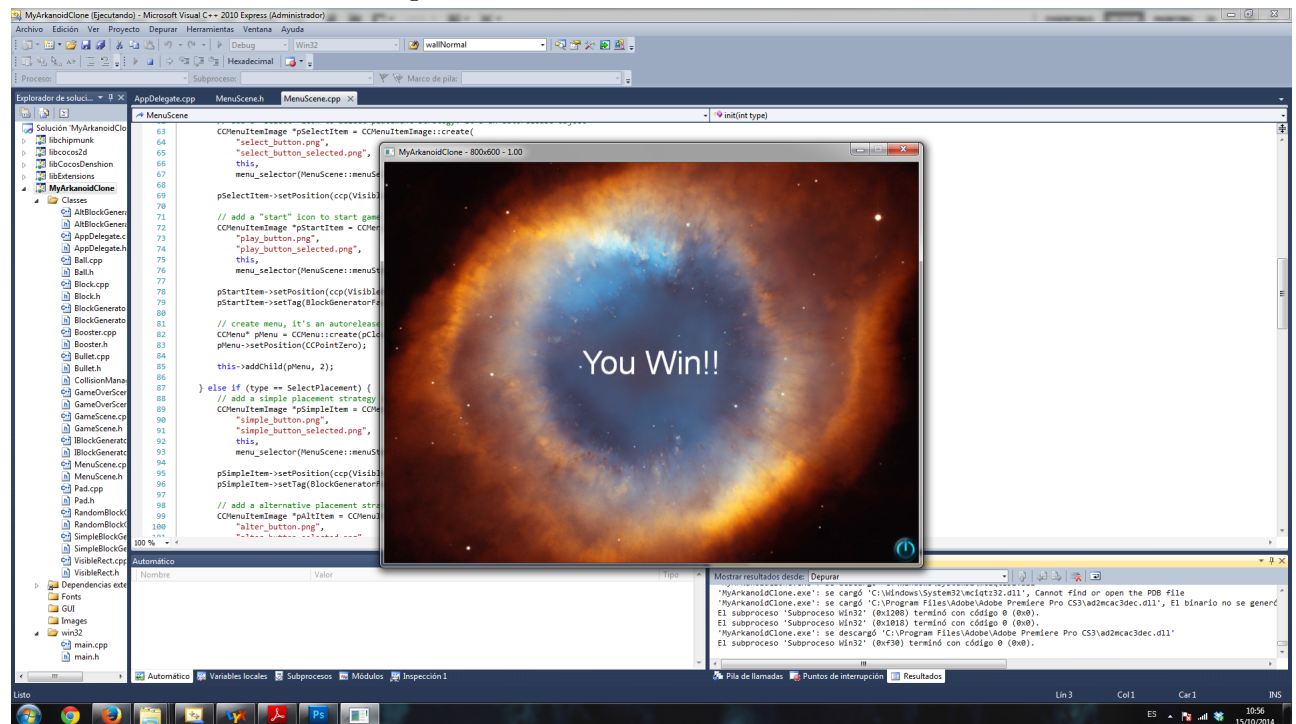
- * these are the various boosters that a block can release, being also subclasses of CCPhysicsSprite, that have an initial downwards impulse (to simulate gravity on a 0-gravity scenario) and which only collide with the pad, making it get a boost (call the pad's *'boost'* method) and destroying the booster (they are also destroyed on colliding with bottom wall). That *'boost'* method gets a type parameter with which the pad chooses the boost case to act accordingly.
- * Life: adds 1 life to the GameScene *'_lives'* counter. The GameScene itself is responsible for polling its pad on the game loop, to see if it has gotten a life boost.
- * Bullet: shapeshifts the pad into the *'laser pad'*, which enables the player to shoot blue energy bullets upwards, by tapping on the pad itself. It is done inside the pad's *'boost'*, when the appropriate parameter is passed, in which case, if the pad isn't already powered with bullet booster, it put its *'_shootingPad'* flag to true, changes its texture to laser pad, and schedules the *'powerUpCounter'* selector.
- * Double: adds 1 extra ball to the game. As in the extra life scenario, the GameScene polls the pad in the game loop to check if it has an extra ball boost, in which case it adds a second ball (tinted in red to distinguish it) to the game scene and to the physics space.
- * Wider: enwidens the pad by 150% during 10 secs. It is done inside the pad's *'boost'*, similarly to the bullet booster case. In this case, if it's not already booster with wider, it put its *'_widerPad'* flag to true, scales itself to 150% in x axis, and schedules the *'powerUpCounter'* selector.

- * that *'powerUpCounter'* selector mentioned above is a callback function called every second, responsible for managing the powered up state timer
- Bullets: 
 - * these are the physics sprites shot by the pad in laser form, which are instantiated from pad's edges and can only collide with blocks, destroying themselves on collision and damaging the blocks with their *'takeDamage'* method.
- Walls:
 - * those are static segment shapes created on the chipmunk physics initialization, on the *'initPhysics'* GameScene's method. They serve for bounding purposes, establishing the 4 game walls, and thus delimiting the game field.
- Collisions:
 - to handle the collision between the various shapes involved in the game, we have defined a set of collision handler functions in the *'setCollisionHandlers'* method of the GameScene. This is the way chipmunk handles physics collisions, where a handler is a set of callbacks between a pair of shapes, which have a collision type defined (those types are set in each sprite init method).
 - those collision callbacks used in the handler are also defined in the GameScene.cpp, and encapsulate the collision logic between a given pair of shapes. They initially retrieve the shapes involved in the collision by their collision type, to do some custom actions according to that collision type.
 - in this case we have almost exclusively used the collision begin functions, as most of the collision logic can be done when the two shapes come into contact. In the cases where we need to do some additional actions (as destroying any of the shapes for example) that cannot be managed in the begin function, what we have done is register a postStep callback with the target shape, where those actions could be done after the physics step (where for example you cannot delete objects).
- Game Loop (*'update'*)
 - the *'update'* schedule selector function represents in this case the main game loop, as it is a function that is called every frame, where the physics step and some other game logic actions are done.
 - specifically, 5 verifications are done inside it, apart from the physics step.
 - * poll the pad to see if it has an extra life or extra ball boost available.
 - * check if any ball goes off-screen, to destroy it if that's the case.
 - * check if there are balls on screen, to update the player lives and reset the regular ball (not the extra one) if there are none.

- * verify the player lives, to see if we lose the game when equal or below 0 lives. In that case we call the sharedDirector to replace the current scene for a GameOverScene, with the type parameter set to GameOverScene::Lose.
- * verify the remaining blocks, to see if we have won the game, when no more blocks are left. In that case, as before, we call the sharedDirector to replace the current scene for a GameOverScene, with the type set to GameOverScene::Win.

Game Over scene:

Figure 0.4: Game Over Scene



-
- it's the ending scene, which has 2 possible outcomes according to a win (no more blocks left) / lose (lives ≤ 0) scenario.
- it is contained in the **GameOverScene** class, which once again, has a 'type' parameter which lets the user create the 2 different scenes (win / lose), with different background image, label, and music.

Comments

- For convenience, ease of debugging and clarity, all the relevant code is commented
- We have used autorelease objects extensively (autorelease pool) to handle memory management.
- Thinking on the easiest parts of the test, I'd say that touch events (easy cocos2d-x support for this) and block life / types has been quite straightforward and easy to implement.
- One of the hardest parts of the test, from my point of view, has been the fact that while I had previous knowledge of C++, I was new to cocos2d and chipmunk environment, and I had to learn it from scratch to develop the game, which is why it took me more time than I expected to finish the test. Also, I had to search several sites, as the cocos2d documentation was quite scarce and most of it was targeted to iOS development and focused on v3.
- Also, the collision management has been a challenge too, as it involved chipmunk learning, and translating from C to C++, etc.