CT30A9300 Code Camp on Communications Engineering 18.6.2010

XNA Code Camp - Summer course 2010

Group 2 aka Team null pointer exception, NinjaGame

Authors: Niko Kurvinen 0312302

Mikko Kaistinen 0334791

Oskar Sonninen 0326770

email: firstname.surname@lut.fi

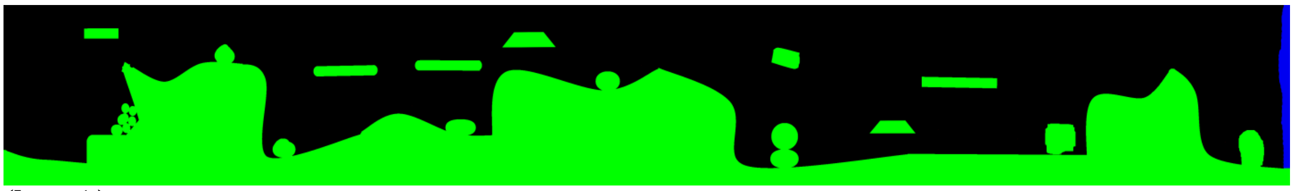
XNA Code camp software description

1. Idea

NinjaGame is a 2D side scrolling platform game. Player character is a black ninja which roams in two different kind of maps. There are two kinds of enemies, red ninjas are walking around and try to prevent player advance on the current situation, they are also lethal. Carnivore plants are immobile and are also lethal. Player mainly tries to avoid enemies, because there are no weapons to eliminate enemies. Player's main tool to avoid enemies is one kind of harpoon called ninja rope, which shoot rope to bottom of upper platforms and so player can climb to higher and move like a Tarzan around the current map. Game graphics are very comic like which looks pleasing and practical. The idea of ninja rope was copied from Finnish 2D platform game called Trine made by Frozenbyte. The architecture of the game enables quick level making so new playable levels are quick to develop with some image manipulation program. User interface is designed to be pretty straight forward and quick to learn. There were few things which weren't implemented to game, because of limited amount of hours in day. There were plans to develop grappling hook which would have worked same as the ninja rope, except it can't grip bottom of the upper platforms. The grand master Judgement day weapon would have been box weapon which was planned to summon huge boxes, which could be used to crush the enemies and get access to higher platforms. It would have been nice to implement those feature to the game, because they would have made game more rewarding to the player. It would have made game more enjoyable, if there would have been more different kind of enemies. The physics engine was also greatly simplified, because of the project time limit.

2. Problems

1. Environment-to-object collision detection was decided to be made with bitmaps where different color represented different material e.g. green represented surface where other objects collide and red dots enemy spawn points (Image 1). XNA framework didn't provide useful functions for this so collision detections for object to environment had to be written from scratch.


(Image 1.)

2. Object-to-object collision was easier to make than environment- to-object collision as XNA had ready to use functions for it.
3. Game levels were designed to be quite large (4000+ pixels wide), but XNA restricted texture sizes to 2048x2048pixels by default.
4. As XNA isn't a game engine, it didn't have any physics simulations so physics needed also to be coded. Co-op of physic, moving and collision detection was really hard to implement as a working solution.
5. When porting the game to Xbox 360 a problem with System.Drawing namespace was encountered as windows platform supported this namespace but Xbox 360 didn't.
6. Game needed some textures, animations and objects.
7. Even game is 2D it was wanted to contain some faked 3D features (depth illusion).

3. Non-technical description

1. Environment-to-object collision based on comparing locations of objects collision points (6 in total per object) and bitmaps color. If objects collision point has the same coordinates as bitmaps green pixel a collision have occurred.
2. Texture size restriction was avoided by loading additional Dynamic-Link Library (http://chrisegner.me/blog/2010/01/large-textures-in-xna/) to XNA that automatically divided larger textures to smaller ones.
3. Object-to-object collision was made with build in function which created an "invisible" sphere around objects and used it as collision edge.
4. As real physics would be awfully hard to code and calculating them real time with nowadays PCs would be impossible physics had to be simplified a lot and final version of in game physics provided basic simulation for velocity, friction and acceleration for game mechanics to work (e.g. swinging around with the hook).
5. Namespace problem with Xbox 360 was solved by importing bitmap as large texture.
6. Objects animations was achieved by making a 3D model in Blender and rendering an animation of 40 frames of that model doing e.g. a walk cycle. After this all images were imported to one .png image as 50x63 pixel images and set in order (Image 2.). Inside the game walk animations is made by simply switching the area to be drawn from the png image.

(Image2)

7. Depth illusion was created with multiple layers e.g. there are 8 different layers in the first game level, that moves at different speed when the screen is scrolling. Scrolling speed for different layers is achieved by making layers different size. Shorter layers scroll slower and wider faster. Also the player move depending on the level somewhere "between" these layer so sometimes the ninja looks like going front a rock and next time from behind.

4. Technical description

Physics

At first, acceleration is calculated; it is combination of gravity, friction and player x-axis acceleration. Gravity is simply a constant vector. Friction is applied only if there is no x-axis acceleration implementing a smooth stop of the player character. Friction is also a constant x-axis deceleration against velocity. The only semi dynamic variable in this equation is x-axis acceleration which is set according to the character's travel direction. Now that the acceleration is known it is added to velocity which is then limited to max velocity. This max velocity has only a limitation for x-component which is adjusted by game pad thumb stick, or if keyboard is used it is assumed to use the maximum velocity. New position is then simply calculated from that velocity. This is all done in Player.Update method shown below in class diagram 1.

Collision detection

A collision with the ground bitmap is checked before accepting the new position information. Ninja object has six collision detection points placed as shown in picture 9674386. Collision of a set of points is detected by interpolating position and new position with one pixel intervals. If collision is detected, that is a Level.MaterialType.WallBit bit is found from the ground bitmap, position preceding that point is returned. First the leftmost or rightmost collision points are checked and one-pixel-before-collision is stored as a new position Next the detection of collisions is divided into two state-based sections, jump and standing/walking. In jump state collision detection section the top collision point, if moving upwards, or bottom collision point, if moving downwards, is used to test collisions with ground bitmap. This is because if the object is on the ground the bottom collision point may be in collide area so it cannot be used if the object have just jumped. Likewise when object have collided with a ceiling it cannot be checked while falling back to ground for ground collision. Those collision points, if any, are used as final new position information. Also if there was a collision while moving downwards state is updated to State.Walking. In standing/walking state collision detection section the y-axis of movement is ignored and only x-axis is used to calculate the new position. First the new position y-component is set so that the slope of movement vector remains as predefined constant. That vector is then checked for collisions and the collide point to downwards is then checked for collisions. The result of these operations is the final new position. If there was no collision found for second collision test (first collide point to down), the object is considered as felled from the edge and its state is updated to State.Jump. This is all done in Ninja.UpdateCollision method shown at the end of this document in class diagram 1.

Scrolling camera

Camera scrolling is very simple. it only needs a current camera position and player x-coordinate to be implemented. it is most easily written in a equation form: camera_position.x += (player_position.x - camera_wants_player_to_be.x) / 24 the only parameter which needs some explanations is camera_wants_player_to_be.x this is actually a constant depending which direction player is moving. It is 1/3 of the screen width if moving right and 2/3 if moving left. the constant 24 is only a result on trial and error. This algorithm smoothly scrolls screen when player moves.

Layer system

Every layer's drawn position is based only to layer's camera position on world map so when camera is half way the bitmap then every layer's mid part is drawn to the screen . Layering system simply works by scaling not the images them self but the position of the image. every image is actually treated as they were the same size, say 1.0, the world width is also scaled from 0.0 to 1.0. Camera position is scaled to world position and that relative position is multiplied with respective layer´s width. That gives the camera position on that layer.

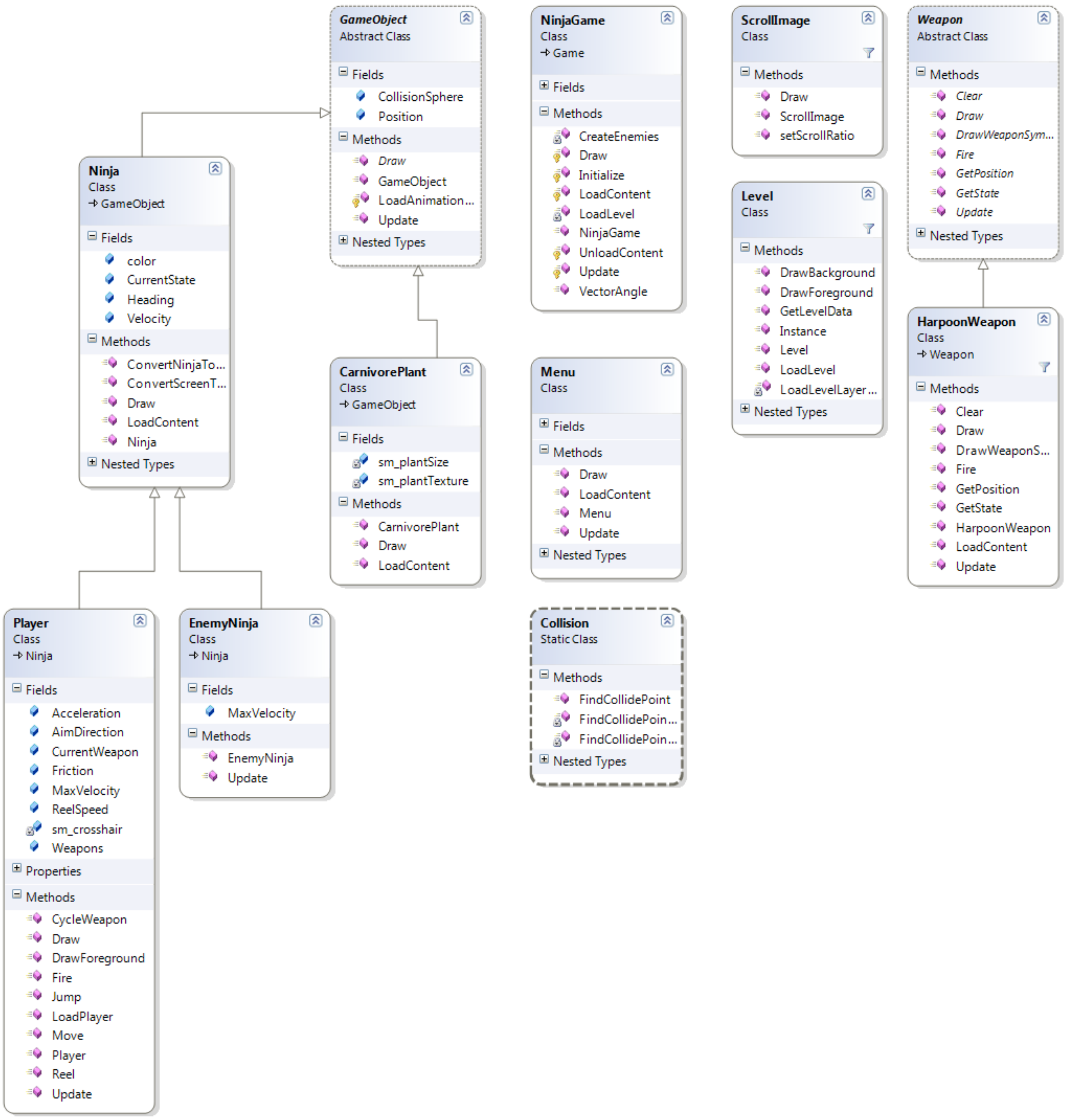
Enemy movement

Every enemy ninja get random walking direction and speed based on their spawning point. Enemy ninjas "patrol route" was done simply following ninjas x-axis velocity. If ninjas previous position is same as new position after calling update method again ninja changes his walking direction walking direction.

5. Involved technologies

Game was created with XNA Game Studio 3.1 framework and Visual Studio 2008 Professional Edition. There were two image manipulation softwares that were used to create game levels; GIMP 2.6 and Paint.NET 3.5.5. Player, some houses and enemy models were made with Blender 2.49b.

Project class diagram:


GameObject
Abstract Class
Fields
CollisionSphere
Position
Methods
Draw
GameObject
LoadAnimation...
Update
Nested Types
NinjaGame
Class
→ Game
Fields
Methods
CreateEnemies
Draw
Initialize
LoadContent
LoadLevel
NinjaGame
UnloadContent
Update
VectorAngle
ScrollImage
Class
Methods
Draw
ScrollImage
setScrollRatio
Weapon
Abstract Class
Methods
Clear
Draw
DrawWeaponSym...
Fire
GetPosition
GetState
Update
Nested Types
Ninja
Class
→ GameObject
Fields
color
CurrentState
Heading
Velocity
Methods
ConvertNinjaTo...
ConvertScreenT...
Draw
LoadContent
Ninja
Nested Types
Level
Class
Methods
DrawBackground
DrawForeground
GetLevelData
Instance
Level
LoadLevel
LoadLevelLayer...
Nested Types
HarpoonWeapon
Class
→ Weapon
Methods
Clear
Draw
DrawWeaponS...
Fire
GetPosition
GetState
HarpoonWeapon
LoadContent
Update
CarnivorePlant
Class
→ GameObject
Fields
sm_plantSize
sm_plantTexture
Methods
CarnivorePlant
Draw
LoadContent
Menu
Class
Fields
Methods
Draw
LoadContent
Menu
Update
Nested Types
Player
Class
→ Ninja
Fields
Acceleration
AimDirection
CurrentWeapon
Friction
MaxVelocity
ReelSpeed
sm_crosshair
Weapons
Properties
Methods
CycleWeapon
Draw
DrawForeground
Fire
Jump
LoadPlayer
Move
Player
Reel
Update
EnemyNinja
Class
→ Ninja
Fields
MaxVelocity
Methods
EnemyNinja
Update
Collision
Static Class
Methods
FindCollidePoint
FindCollidePoin...
FindCollidePoin...
Nested Types