CS4344 Project Report - Nutty Ninjas X

Proudly Presented By:

Name	Github	Matric Number	Email
Tay Yang Shun	yangshun	A0073063M	tay.yang.shun@gmail.com
Ng Zheng Han	NgZhengHan	A0086934R	ng.zheng.han@gmail.com
Nguyen Trung Hieu	ngtrhieu	A0088498A	ngtrhieu0011@gmail.com

Game Design and Implementation

Game Description

Genre	2D Side Scroller, Platformer, Shooter, MOBA
Game Elements	Players, Shurikens, Portals
Game Content	Ninjas don't actually have teleporting abilities. They leverage on technology by creating futuristic inter-spatial portals which allow objects to travel through them. Nutty Ninjas X is a sequel to the hit game Nutty Ninjas, and it makes use of this portal mechanics to add a new twist to the original game. In this fast-paced sequel, ninjas battle against each other in a test of wits and reaction skills by eliminating as many opponents as they can while trying to stay alive.
Theme	Retro Japanese
Number of Players	Real-time Multiplayer

Game Reference

Reference	Nutty Ninjas, Portal 2
Art Reference	Stick Hero, Patapon

Game Technical

Technical Form	2D graphics
View	Side Scroller
Platform	HTML5 Web
Supported Device	PC and Mobile Phones
Game Engine	Quintus, which is an easy-to-learn, fun-to-use JavaScript HTML5 game engine for mobile and desktop. Out of the box, Quintus provides many of components required to build a game, such as animation, physics engine, audio management, asset loading, etc. URL: http://www.html5quintus.com/

Implementation Details

Modular Implementation

The implementation heavily makes uses of modules/packages in order to make the game extensible and maintainable. There are 3 kinds of modules being used: (i) Node Modules, (ii) Bower Packages, (iii) JavaScript Modules.

(i) NPM (Node Package Manager) is the package dependency manager for Node.js libraries and an increasing number of JavaScript libraries and frameworks are being published on NPM and can be easily installed via the command line. We used Express and Socket.io libraries.

(ii) Bower is similar to NPM mentioned earlier, but it is meant for front end libraries. Using a package manager for libraries has the benefits of easy updating of the libraries being used in the project and that these libraries do not have to be checked into version control, which keeps the commit history clean. Using bower, we installed front end JavaScript libraries such as PubSub, jQuery, AngularJS.

(iii) JavaScript Modules are essentially as JavaScript objects/functions. We chose to save each module in a separate file to make the source code organised and structured. Working in a team, many people will be modifying code simultaneously. Hence it will be easy to compare changes made by our teammates if each module was contained in a separate file. Quintus provides an OOP-style way of programming game classes and made heavy use of it to reuse and reduce the written code. For example, the Actor class represents a ninja that is present in the screen. The Player class subclasses the Actor class and use controls are added to it. AngularJS was used to update the interface for the lobby, scoreboard and ammo count. Each of these components were saved as a separate controller function within the scripts file.

Building Interface With DOM

Because Quintus' support for in-game UI elements isn't that fantastic, and is quite limited, we only used HTML5 canvas for the rendering of the game. The lobby screens, and UI elements present in the game were rendered using HTML5 DOM rather than canvas. This is because canvas does not render elements in high resolution on a retina display screen. Since the UI elements do not change often, and does not heavily rely on the game engine, we opted to use HTML5 DOM to render these elements instead. These DOM elements were updated using AngularJS, which provides two-way binding of the interface and data models. It is easy to build the lobby screen interface with DOM than with canvas using Quintus.

Assets Loading

Assets (graphics, levels, music) are loaded before the game starts. Quintus handles asset loading very well and only starts the game after all assets have been loaded.

Socket Communication

Socket.io supports Rooms and Namespacing. Lobby and matchmaking-related socket events will under the `/lobby` namespace while game-related events will fall under the `/game` namespace. Hence multiple concurrent game sessions are supported.

(i) Lobby and Matchmaking-related events

Most of the server side logic is housed within `room-manager` directory. If a player joins a room with a name that does not exist, a room with that name will be created. Else, players will join that existing room.

We created a lobby screen showing how many games are present and how many players present in each room. This lobby screen can be accessed via the /lobby route. In this screen, players can create a new room or choose a room to join. They just have to add in their name and select a colour. We support unlimited number of rooms and players, with the only limitation being the server load. However, when there are too many players in the game, the games becomes very messy. It is advised that there should not be 8 players in a room at any point of time.

The lobby interface is being built using HTML5 DOM rather than canvas for reasons mentioned earlier. The code can be found in `scripts/LobbyController.js`. The lobby page listens for the “lobby.state” event that is being published whenever a user joins a game. Whenever the lobby page receives a “lobby.state” event, the interface is automatically refreshed with the new state from the server using AngularJS.

(ii) Game-related events

Using Socket.io rooms functionality, we could effectively separate out the communication relevant to each room. Sending a message to a room is simple using one line of code:

```
socket.broadcast.to(roomId).emit('player.disconnected', player.getState());
```

Hence game-related events are restricted to the room that the players are present in, and not received by all existing games running on the server.

Mobile Support

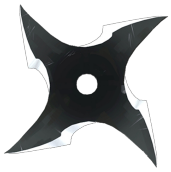
Besides the touch sensors, when played on mobile, the device’s accelerometer is used as an additional input. Users can shake the device as a convenient way of toggling between the shooting modes. The code listens for the `devicemotion` event which returns the current acceleration of the device. If the combined acceleration of the event in three axes is above a certain threshold, the weapon will be toggled.

Additionally, on mobile, if a player is being hit by a shuriken, we provide tactile feedback via vibrations. Note that not all mobile browsers support this behaviour (as of present, only mobile Chrome and Firefox).

Unique Stats for Ninja

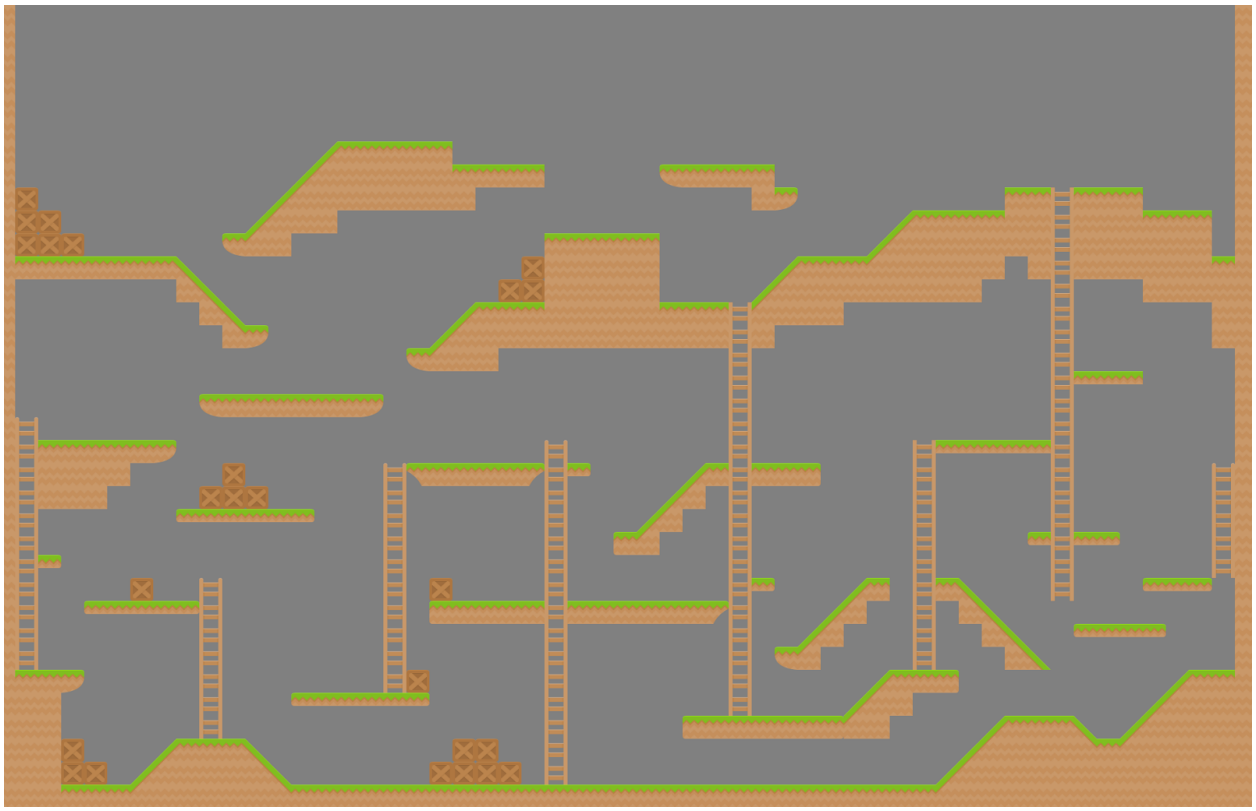
There are 4 different kinds of Ninjas: Fire/Red, Aqua/Blue, Leaf/Green, Lightning/Yellow. Each ninja has special characteristic, such as being able to jump higher, move faster, fire shurikens at a faster rate, etc. All these can be configured within the file `scripts/Game/Constants.js`. This enforces separation of data and logic, and the team member in charge of game balancing only has to look into that file to tweak the values for each ninja.

Gameplay Entities

Ninjas 	Ninjas can move and jump around using the **WASD** buttons corresponding to: Up, Left, Down, Right respectively. Ninjas can fire two types of projectiles: Shurikens and Portals. The **E** button can be used to toggle between Shuriken mode or Portal mode. On the mobile, there is a virtual joystick that can be used. Players have a fixed amount of HP and when they die, they respawn at a random location on the map.
Shurikens 	Shurikens can be fired in any direction using the mouse cursor. They are able to bounce off surfaces and reflect off other projectiles. When shurikens hit ninjas, they deal damage to the ninjas. Each ninja starts a fixed amount of shuriken in their inventory and they use them up when they throw shurikens. Their inventory regenerates over time and their shuriken stash refills. This is to prevent mindless spamming of weapons.
Portals 	Portals have the ability to teleport objects. Both players and shurikens can be teleported between portals. Each player can only have a pair of portals existing in the game at any one time. Subsequent portals fired will replace existing created portals. If there is only one portal existing in the game, then the portal does not teleport objects.

Level Design

As Nutty Ninjas X is a MOBA (Multiplayer Online Battle Arena), our map had to feel like an arena with infinite gameplay possibilities. Hence unlike typical levels found in platformer games, which are mostly linear, we created a level that had no dead ends and there are many ways to get from one location to another. A screenshot of the whole level is shown below.



The reason for the relatively large size of the map is that players can easily create portals to teleport between locations instantly. Hence with the use of portals, they can travel from end of the map to another in an instant. This creates the possibility of a player leaving a portal at one quiet corner of the map, going into a battle, creating a portal in the battle, and escape back into the quiet corner if he is dying, and close the portal after that.

Design and Implementation of Networking

Which type of architecture or communication model did you use?

We used a distributed hybrid architecture consisting of smart clients and dumb server.

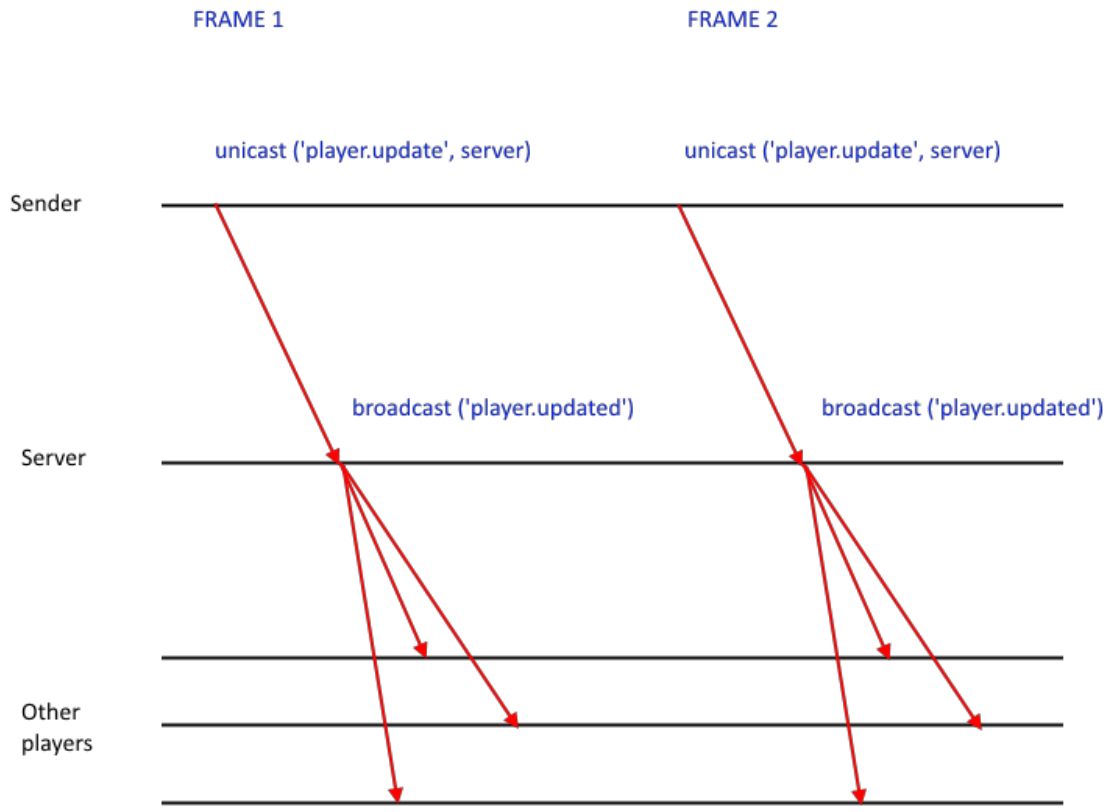
Dumb Server - The server is in charge of relaying messages between clients and also measuring and keeping track of the latency of each client. The server also adopts an simple distance-based interest management scheme, and only sends updates to players if they are within a certain distance from each other.

Smart Client - Each client will simulate the game simultaneously and be in-charge of interactions related to them, such as their character as being hit by projectiles. Each character's information and non-movement-related player-issued commands are sent to the server and the server broadcasts these information to other clients.

How did you synchronize the states among the players?

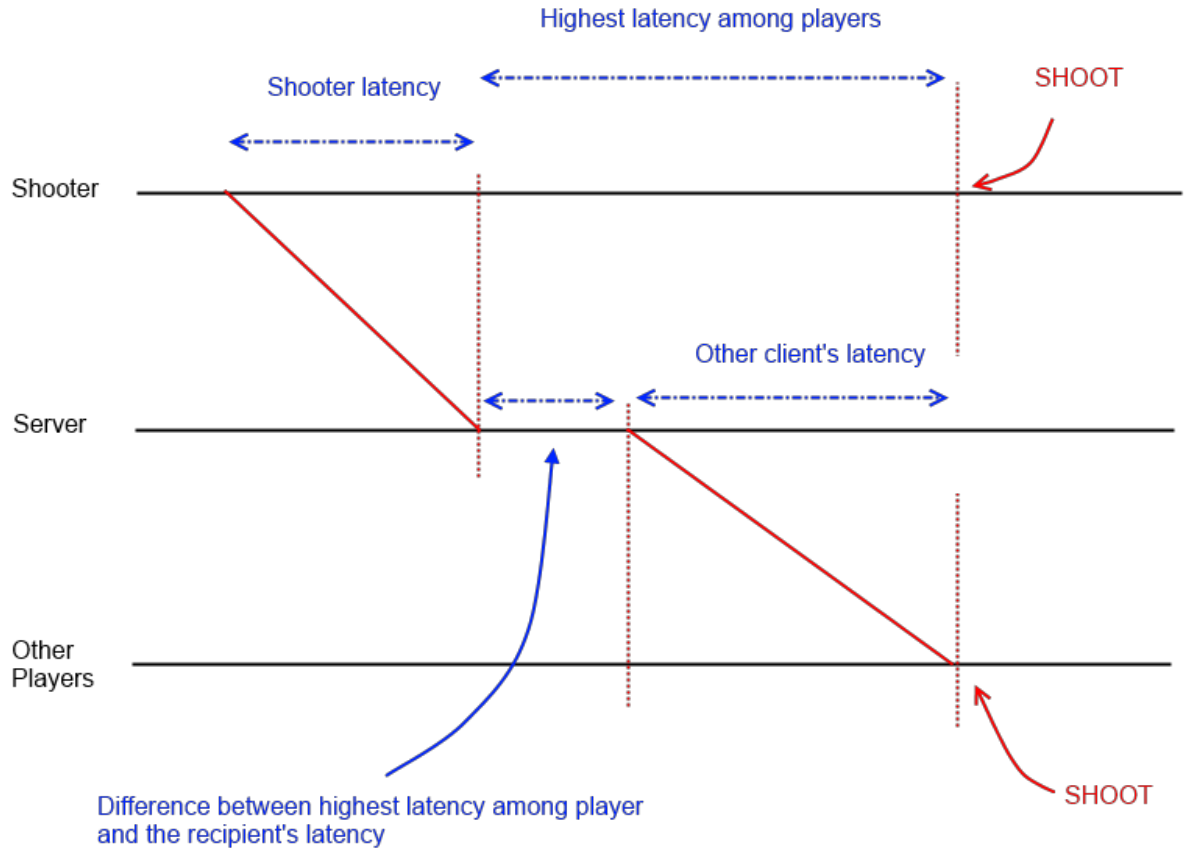
We used lag compensation techniques for the entities such as Actors, Shurikens and Portals:

Actors - Actor are avatars controlled by other players. As there is virtually no way the client can predict the movement of the actors, we synchronizing player's movement by letting each client constantly sending update of the player's state to the server. Server in turn will broadcast the state to all other clients. The clients upon receiving update event from server will immediately reflect to the corresponding actor.
We didn't perform local lag on actors because actors are slow moving objects in the game, hence the error would be unnoticeable.



Shurikens - Shurikens are moving entities and it is important that shuriken positions are consistent between players for them to react correctly. Since shuriken movement can be easily predicted by simulating physics, client only sends shuriken fire event to server, which in turn will be broadcast to all other clients. Other clients will render the shuriken as soon as they receive the message.

However, since there is no further synchronization message after fire event, it is important to all clients to create the shurikens at roughly the same time. In order to achieve this, we employ local lag on the client who sends the fire event. Local lag is calculated and implemented as follow:



As shown in the diagram above, client who wants to shoot shuriken first must fire an ‘player.shoot’ event to the server. Then it would need to wait for total of its own latency plus the highest latencies among all players. The server upon receiving the shoot event would need to broadcast to all other clients. However, server will need to employ local lag for each recipient, delaying sending the message for amount equal to the difference between the highest latency among all players and the latency to the recipient. By using this model, we ensure that all clients would receive the shoot event roughly around the same time, regardless of latencies to server.

Portals - As the presence of portals will affect the core gameplay, it is important that portals appear on every client’s screen at the same time. Server messages to show the portals are not sent at the same time to each client; artificial delay is introduced depending on the the latency of the client. Messages to high latency clients will be sent earlier as compared to messages to low latency clients, so that portals will appear on clients’ screen roughly at the same time.

Aside those techniques, we have also considered implementing Local Perception Filter (LPF) for Shurikens, and we, in fact, did implement Local Perception Filter during development. However, the implementation was removed due to the changing the gameplay mechanics. Although it is an effective implementation, since shurikens can collide with each other midair, making prediction of the hit position virtually impossible. Moreover, since each clients simulating the game independently, LPF poses as a potential problem for breaking state synchronization between clients. Therefore, we all agreed that LPF would be removed in the final version of the game.

What strategy did you use to reduce the bandwidth / power usage of the game?

As there will be many projectiles in the game, it is not efficient for the server to update each client about the position of each projectile. Projectiles in the game have a fixed movement pattern and hence their movement are predictable and deterministic. Therefore only the creation of the projectile (starting position and velocity) are communicated to clients.

Furthermore, simple **distance-based interest management** is employed to further reduce the bandwidth needed. In the game, only the Actors requires constant synchronization and these synchronizations place high stresses on server. Therefore, interest management only focuses to reduce the size of the packets and the number of packets sent.

In order to achieve this, we split the packet payload into 3 sections: important, compulsory, optional. Important fields are fields that need to be delivered to all other clients. In the game, these fields are: player's name, player's hp are player's color. Packet most of the name doesn't contain these fields, and they are only added to the packet by the client when changed. Packets with important fields are marked as important. Compulsory fields are fields that always presents in the packets. They are namely, playerId and player coordinate (x and y). Lastly, Optionals fields are not important fields that can be absent from the update packet. Player velocity and facing direction are among those

The server without interest management, upon receiving the update packet from a particular client, would broadcast to all other clients. However, with interest management, it will only forward the packet to selective clients. The selection process is described below:

Server will calculate distance between the player's avatar and the others. Only those who are close to the sender are forwarded the full packet. Clients that are at medium distance will only have 70% chance to receive the packet. Also, the packet will also suffer 50% chance to have all their optional fields removed. Clients that are too far away will only be updated 30% of anytime, with all optional fields removed.

If the packet is important (i.e: containing important fields), those will be sent to every other clients in the game (i.e: will not be subjected to randomness in skipping/sending the packet). However, optional fields in the packet may still be removed.

This scheme can help the server reduces both the number of packets sent and the size of each packet, by sacrificing the information quality of the packet.

In the game, we allow new players to join any room at any time. This normally would require the server to send game states of all players, shurikens and portals that are in the game to the newly joined player, and hence would cause the welcoming packet large. We resolve this problem by employing 2 strategies. Firstly, we only send other player's important informations (as described above: playerId, name, hp and color). The rest of the player's status would be updated by normal updates packet. Secondly, we do not send information about shurikens and portals that are currently in the game. In order to avoid newly players from getting hit by 'unknown' shurikens, we purposely design the maps so that spawning points would be far away from the center areas where players would likely to battle. Hence, the newly joined players can use the time travelling to the center to actually get updated for new shurikens and portals. By the time the newly joined player arrives, all 'unknown' shurikens and portals should already expired.

How did you ensure fairness among the players?

We ensure the fairness among the players by keeping the game state as consistent as possible.

Summary of Networking Protocols Used

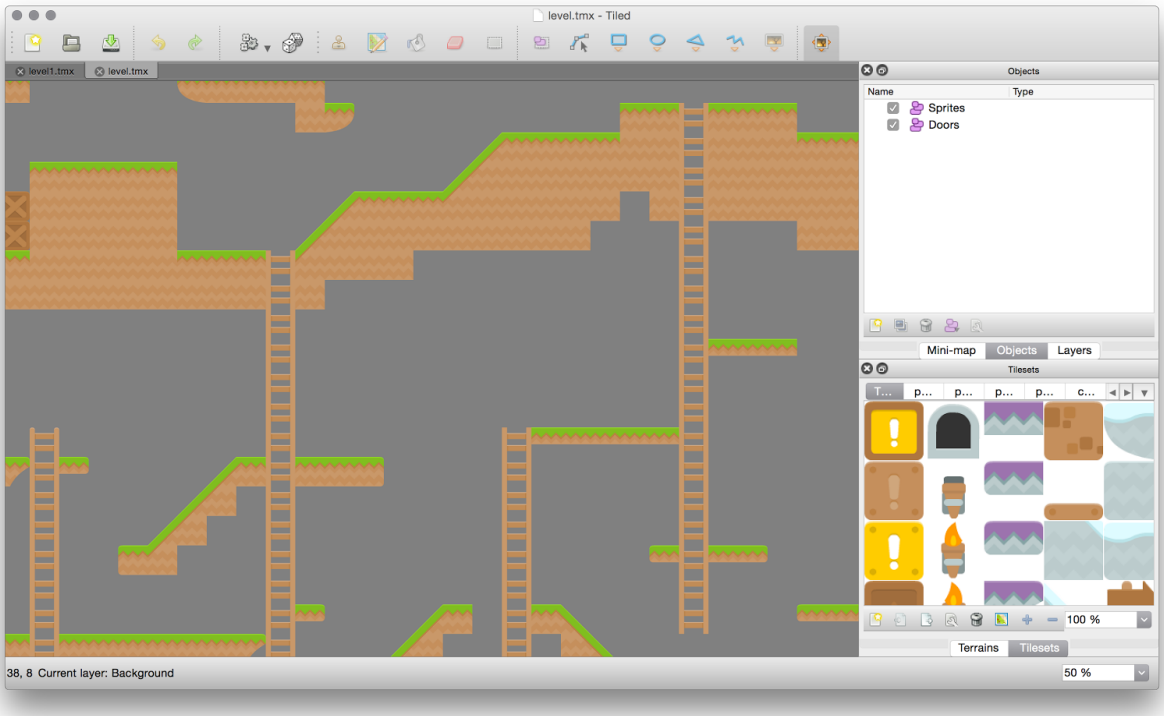
- Latency Measurement
    - The server pings each client every X number of seconds to measure the latency. These latency values may be used in the subsequent policies mentioned below.
- Lag Compensation
    - **Shurikens** - Shurikens are moving entities and it is important that shuriken positions are consistent between players. Server messages to add shurikens are sent at the same to each client and the client renders the shuriken as soon as they receive the message. However, to account for differing latencies, the server predicts the position of the shuriken when the client receives the message by using the shuriken velocity and client latency.
    - **Portals** - As the presence of portals will affect the core gameplay, it is important that portals appear on every client's screen at the same time. Server messages to show the portals are not sent at the same time to each client; artificial delay is introduced depending on the the latency of the client. Messages to high latency clients will be sent earlier as compared to messages to low latency clients, so that portals will appear on clients' screen at roughly the same time.
- Distance-based interest management
    - Players only get updates of coordinates opponents that are in the same screen as the player.
    - Players still get updates of actions of all opponents for shurikens, portals, damage.

Asset and Artwork

Not too much time was spent on creating assets and artwork as it is not the core focus of the project. However, we did not use simple graphics for the art as a minimalist art style was not suitable for a flashy platformer game involving ninjas. Hence we made use of some existing game art assets and modified them intelligently to suit our needs. For example, the player's original sprite asset was a green alien that looked like this:

We used photoshop to modify it to make it look like:

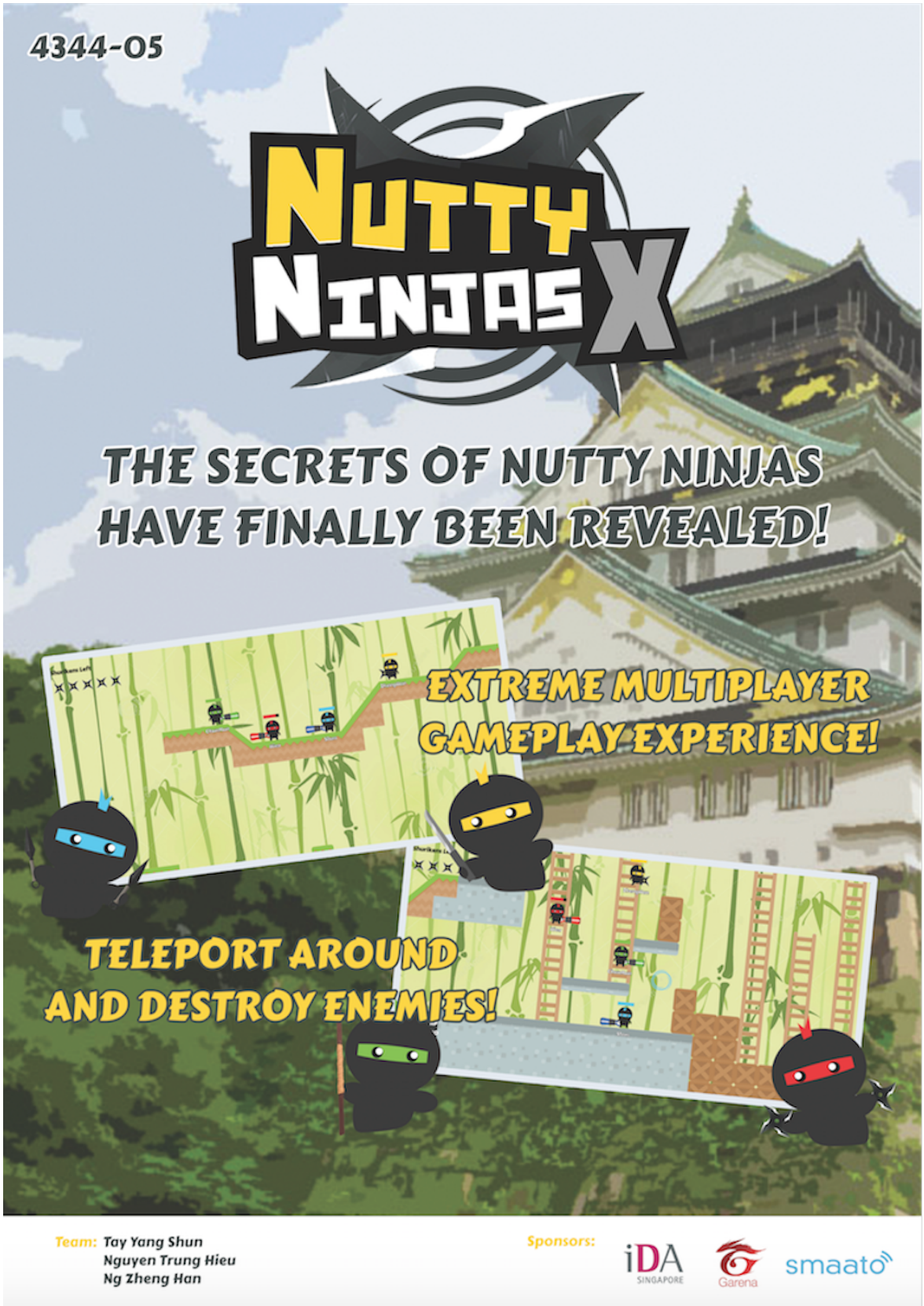
Level were saved in the TMX format and TMX editors were used to create the level easily.



Trailer Video

Our video/trailer can be found at the following URL: https://www.youtube.com/watch?v=0mEBeMMJwkY

Poster


4344-05
NUTTY NINJAS X
THE SECRETS OF NUTTY NINJAS HAVE FINALLY BEEN REVEALED!
EXTREME MULTIPLAYER GAMEPLAY EXPERIENCE!
TELEPORT AROUND AND DESTROY ENEMIES!
Team: Tay Yang Shun
Nguyen Trung Hieu
Ng Zheng Han
Sponsors:
iDA SINGAPORE
Garena
smaato