Progetto di Interactive Graphics

DragonSnake21

Michele Ciciolla
1869990
https://github.com/micheleciciolla

Flavio Lorenzi
1662963
https://github.com/FlavioLorenzi

Francesco Cassini
785771
https://github.com/francesco-AI

Matteo Ginesi
1832198
https://github.com/matginesi

Prof: Schaerf

AA: 2018 - 2019

Course: Interactive Graphics - Spring 2019

Indice

Capitolo 1

Breve presentazione del lavoro

L'applicazione qui descritta è stata pensata come un riadattamento del celebre gioco *Snake*[1] in chiave tridimensionale ed arricchendone l'ambiente con nuove texture ed ulteriori personaggi.

Tutto il team ci tiene a sottolineare che il lavoro è stato costruito da zero in tutte le sue parti: i quattro modelli gerarchici e le relative animazioni sono state curate componente per componente.

In base alla scelta dell'ambiente in cui giocare sono stati creati tre ambienti ben distinti, con in comune solo la struttura dei modelli sopra citata. Per ognuna di essere infatti cambiano texture e musica.

Il funzionamento si basa sull'interazione dell'utente mediante la tastiera: comandi `W,A,S,D` per il movimento planare ed `Z,X` per quello spaziale.

Lo scopo del gioco è quello di sopravvivere il più possibile cibandosi dei target che sono presenti sulla mappa: **un uovo, una pecora ed una papera**. All'aumentare dello score raggiunto, aumenta anche la velocità di movimento, in modo da rendere la difficoltà di gioco più elevata. Oltrepassando la mappa di gioco il gioco termina con un **game over**.

[1] Nokia®

Capitolo 2

Ambiente di sviluppo ed aspetti tecnici

Il lavoro è stato pensato ed implementato partendo dalle linee guida su `WebGl` offerte durante il corso ampliandone il ventaglio di strumenti grazie alla ben più ricca libreria `THREE.js`.

2.1 `THREE.js` e WebGL

Il nostro gioco è basato principalmente su due processi: una pagina `HTML` di intro col titolo ed i bottoni per l'iterazione con l'utente, tramite i quali si ha la possibilità di scegliere un ambiente di gioco; il gioco vero e proprio implementato in `javascript` utilizzando la libreria `THREE.js` basata su `WebGl`.

2.1.1 `HTML`

La sezione `head` dello script `HTML` introduce il *canvas* in cui è fissato la background image, gli input specifici per i tre bottoni a cui sono collegate le classi `alignleft` e `alignright` che permettono di agire in modo univoco su di essi.

Tutti i titoli seguono lo stile `CSS` di una delle due classi `gametitle` e `gamevariant`. Inoltre ogni `ID` è specificato con # includendo la sua posizione nello spazio e visibilità iniziale.

Un esempio di bottone è quello associato alla scelta di ambientare il gioco su Marte *(The Red Planet)* per cui l'evento "click" permette il passaggio alla schermata di gioco associata; cosi proposto:

```
<button class="link alignleft" style="font-size:3em;" id="Mars">
The Red Planet</button>

<img src="./textures/intro-mars.png"
style="position:absolute;top:18.0em;left:4.1%;width:20%;
"id="intro-mars">
```

Figura 2.1: Pulsante per la scelta dell'ambiente di gioco "The Red Planet".

La scelta dell'utente è registrata grazie alla funzione `document.getElementById().onclick` presente sullo script `app.js`, la quale legge l'`ID` specificato e avvia la procedura al suo interno.

2.1.2 JavaScript

Dal lato `JavaScript` a `WebGl` è stato delegato il background del lavoro: è infatti possibile notare nello script `app.js` come gli oggetti `camera`, `renderer` e tutto il ciclo di animazione si basi su tale libreria.

Di fondamentale utilizzo sono i metodi `setTitle()` per la gestione delle scritte su schermo, `main()` per la creazione dell'oggetto `gioco`[1], `game.music` per la scelta dell'audio da riprodurre[2].

`THREE.js` è stato di fondamentale utilità nella creazione dell'oggetto `scene` (`new THREE.Scene()`), nei componenti dei modelli protagonisti del gioco e delle interazioni con l'utente grazie alla libreria `OrbitControls`.

2.2 Oggetti *Map* e Game

Come specificato nel paragrafo precedente, l'oggetto `Game` è la colonna portante del lavoro, esso viene costruito mediante una chiamata `new Game` che successivamente crea la scena, la telecamera, i controlli utente e l'oggetto renderer.

Associata a tali procedure vi è la creazione della mappa di gioco da parte della classe `Map` che sfrutta la libreria `THREE.PlaneGeometry` per creare il campo da gioco e le relative caratteristiche materiali. Il processo seguente aggiunge gli oggetti luce (`game.addLights()`) di tre tipologie:

- `THREE.SpotLight`: punto di luce.
- `THREE.HemisphereLight`: semisfera di luce ambientale che copre l'intera scena.
- `THREE.DirectionalLight`: luce diretta, responsabile della maggior parte delle ombre.

Le tre sorgenti di luce garantiscono un buon effetto visivo di ombre e riflessi. L'inizializzazione del gioco si conclude con l'aggiunta delle texture e dei modelli di gioco ai quali è dedicato un paragrafo più in basso.

2.3 Textures

Una texture è un'immagine bidimensionale riprodotta su una o più facce di un modello poligonale tridimensionale. Per arricchire il gioco e renderlo di qualità abbiamo utilizzato una vasta quantità di immagini, a seconda del livello/mondo in cui ci si trova. Gli script in cui vengono gestite sono:

- `app.js`: scena e mondo.
- `snake.js`: drago.

I formati delle immagini trattate sono tutti `jpg` e `png`. Come abbiamo visto in `app.js` è presente la variabile globale `selectWorld` la quale, a seconda del suo valore, permette un rendering grafico diverso. Qui abbiamo quattro diverse funzioni:

```
if (textureAttive)
{
  game.createRiver(5, 0, 50, -0.4, 0);
  game.createFloorSx(20, 0, 50, 12.5, -0.4, 0);
  game.createFloorDx(20, 0, 50, -12.5, -0.4, 0);
  game.createSkyBox();
}
```

Le funzioni `createRiver(), createFloorSx(), createFloorDx()` si occupano della "pedana" di gioco e per ogni mondo applicano le texture associate, già precedentemente caricate in memoria: in una variabile di tipo `THREE.Geometry` è definito un `THREE.BoxGeometry` di tre dimensioni tramite il quale viene chiamato il metodo `applyTexture` presente in `utils.js` in cui grazie alla funzione `THREE.ImageUtils.loadTexture` viene caricata, salvata in memoria e gestita l'immagine da applicare come texture alla quale viene successivamente definita la "location" di applicazione; questa è mappata nella variabile `Material` grazie al metodo `THREE.MeshBasicMaterial`. Infine, come appreso a lezione, viene effettuato il **Mesh** tra `Geometry` e `Material`. Questa nuovo oggetto `Mesh` viene aggiunto alla scena[3].

Il metodo `createSkyBox()` agisce in maniera simile, ma poiché si occupa di allestire il mondo esterno, attraverso `THREE.ImageUtils.loadTextureCube(urls)` carica sei diverse texture (una per ogni lato interno del cubo di scena: *front, back, right, left, up* e *down*) creando una `THREE.BoxGeometry` (geometria a cubo) la quale fa da "contorno" alla scena totale, di dimensione[4] $4096 \times 4096 \times 4096$. L'oggetto aggiunto alla scena sarà quindi:

[1] Nello script `app.js` oltre alla definizione della classe `Game`, all'interno della funzione `main()` avviene l'allocazione, quindi la creazione, dell'oggetto e la definizione delle varie funzioni di update associate ad esso.
[2] Delegando la gestione del buffer audio alle funzioni implementate dalla suddetta libreria `THREE.js`.
[3] Proprietà della classe `Game` inizializzata nel `main()`.
[4] Dimensioni espresse nelle coordinate (x, y, z) relative all'inquadratura, quindi non assolute.

```
sky = new THREE.Mesh(new THREE.BoxGeometry(4096, 4096, 4096),
skyMaterial)
```

caratterizzato da un **Mesh** tra la `Geometry` descritta sopra e una variabile `Material` definita come oggetto `THREE.ShaderMaterial`.

Prendendo ad esempio `selectWorld = 1` (*Mars level*) avremo:

```
river = "textures/mars/river.jpg";
floor = "textures/mars/floor.jpg";
directory = "textures/mars/";
satellite = "textures/mars/sat.png";
```

L'oggetto "satellite" in questo caso è un semplice rendering di un'immagine `png` utilizzata per arricchire la scena, con la particolarità che anche ruotando la telecamera essa si muove di conseguenza rivolgendosi sempre verso il nostro **point-of-view**: ciò è garantito da `THREE.Sprite(satMaterial)`.

Figura 2.2: Diverse texture del campo di gioco e del drago, nelle diverse ambientazioni.

In `snake.js` invece ci siamo occupati di arricchire "head" e "body" del protagonista grazie alla funzione:

```
function chooseTexture() {
if (selectWorld == 0) {
  // land
  skinFile = 'textures/land/skin.jpg';
  headFile = 'textures/land/head.jpg'; }
if (selectWorld == 1) {
  // mars
  skinFile = "textures/mars/skin.jpg";
  headFile = "textures/mars/head.jpg"; }
if (selectWorld == 2) {
  // dark
  skinFile = "textures/dark/skin.jpg";
  headFile = "textures/dark/head.jpg"; } }
```

A seconda del valore della variabile `selectWorld` cambierà il suo look[5]. Qui ogni texture è caricata in modo semplice così:

```
var bodyTexture = new THREE.TextureLoader().load(skinFile);
var materiale = new THREE.MeshBasicMaterial({
  map: bodyTexture });
var blockMesh = new THREE.Mesh(this.blockGeometry, materiale);
```

con le relative proprietà `castShadow` e `receiveShadow` settate a `true`.

[5] In relazione all'indirizzo dell'immagine `jpg` passata.

Capitolo 3

Modelli implementati: Low poly style

Tutti i modelli implementati sono stati creati dal team e non scaricati da internet. Ai fini del gioco, cioè quello di catturare degli elementi, abbiamo creato i seguenti modelli:

- il drago con una testa completamente animata
- un uovo come cibo per il drago
- una papera sempre come cibo per il drago
- un capretta

Per quanto riguarda il paesaggio sono stati creati:

- le nuvole che si muovono sullo sfondo
- un albero "normale": un pino che agita le fronte
- un cactus
- una palma

Anche questi modelli sono stati creati internamente ed animati "frame a frame", tramite l'utilizzo di una variabile temporale.

3.1 Il Drago

Il drago è costituito da due parti separate che sono animate in modo diverso. La testa e la coda infatti sono due gruppi separati tenuti attraverso una variabile `snakeGroup`; mentre per la testa è stato creato un nuovo gruppo gerarchico al fine di realizzare una serie di movimenti relativi solo a questa parte del corpo.

La struttura gerarchica della testa è la più complessa rispetto a tutti gli altri modelli complessi. Infatti conta oltre 40 pezzi organizzati gerarchicamente: ad esempio la bocca è costituita da 2 parti ed ogni parte ne contiene a sua volta oltre venti.

Figura 3.1: Faccia del drago.

3.1.1 Classe `Snake`

```
class Snake {
  constructor(selectWorld) {
  this.selectWorld = selectWorld;
  chooseTexture();
  this.snakeLenght = 0;
  this.snakePosition = new THREE.Vector3(0, 2, 0);
  this.snakeRotation = new THREE.Vector3(0, 0, 0);
  this.snakeGroup = new THREE.Group();
  this.snakeGroup.name = "SnakeGroup";
```

3.1.2 Select world per le TEXTURE

La classe `Snake` che contiene tutti i pezzi di cui è composta la testa: come si vede contiene anche una variabile che in base al mondo selezionato può far variare la texture: sono state previste tre diverse textures a seconda del mondo che è stato scelto.

```
const head = new THREE.Group();
head.position.set(0, 0.61, 0.5);
head.rotation.set(0, 0, 0);
this.snakeGroup.add(head);

var headTexture = new THREE.TextureLoader().load(headFile);
this.skinMaterial = new THREE.MeshLambertMaterial({ map: headTexture});
if (!textureAttive) this.skinMaterial = new THREE.MeshBasicMaterial({ color: Math.random() * 0x00
      });
```

3.1.3 Funzione `AddBlock`: il drago si allunga dinamicamente

Per quanto riguarda i blocchi che compongono il drago sono realizzati sempre attraverso l'uso di blocchi geometrici di `THREE.js`. La particolarità sta che il corpo non è un corpo statico, ma è stata creata una funzione che aggiunge ulteriori componenti al corpo.

```
addBlock() {
  var bodyTexture = new THREE.TextureLoader().load(skinFile);
  var materiale = new THREE.MeshBasicMaterial({ map: bodyTexture  });
  var blockMesh = new THREE.Mesh(this.blockGeometry, materiale);
  blockMesh.castShadow = true;
  blockMesh.receiveShadow = true;
  blockMesh.position.z = - (1.1 * this.blocks);
  blockMesh.name = "Snake:Tail_" + this.blocks;
  this.snakeGroup.add(blockMesh);
  this.blocks++;
}
```

3.1.4 Movimento: i *6 gradi di libertà* del drago, 6 tasti per la direzione

Per quanto riguarda il movimento del drago, sono state create diverse routine. Innanzitutto è stato necessario separare il movimento della coda da quello della testa. C'è un movimento generale di tutto il gruppo `Snake` ed ogni blocco che segue la testa si deve riallineare ad ogni cambio di movimento. La complessità è data dal numero di possibili mosse che il giocatore può compiere: sono ben 6 le possibili direzioni che il drago può intraprendere e quindi è stato necessario introdurre sia una funzione in grado di catturare l'input del giocatore.

```
    // [ Z ]
    case (90):
      if (this.isEqual(this.snakeDirection, [0, -1, 0])) {
      console.log("Auto eat");
      if (this.length == 0)
        this.isDead = true;
      break;
```

Ogni volta che la direzione viene cambiata, tutte le parti del drago vengono orientate nuovamente in base alla nuova direzione. Mentre la funzione `setOrientation()` tiene conto del verso del movimento, le due funzioni `moveHead()` e `updateBody()` ci aiutano a capire come le due parti del drago, testa e coda, agiscano insieme, ma vengano aggiornate separatamente. Le funzioni contengono difficoltà diverse.

Infatti la funzione `updateBody()`, anche se deve aggiornare la posizione dei semplici blocchi, deve ricevere in ingresso la posizione di ogni blocco precedente al fine di far percorrere a tutti lo stesso percorso.

3.1.5 `moveHead()`

Il movimento della testa è il più complesso e completo presente nel gioco. Aiutandosi con la struttura gerarchica del modello testa, le parti in movimento nella testa sono numerose.

```
this.snakeGroup.children[0].children[8].rotation.x += -0.02;  //mouth UP
this.snakeGroup.children[0].children[9].rotation.x += 0.02;   //mouth DOWN
this.snakeGroup.children[0].children[1].rotation.x += 0.03;   //EAR RIGHT
this.snakeGroup.children[0].children[2].rotation.x += -0.03;  //EAR LEFT
this.snakeGroup.children[0].children[5].rotation.y += 0.04;   //SQUAME 1
this.snakeGroup.children[0].children[6].rotation.y += -0.04;  //SQUAME 2
this.snakeGroup.children[0].children[7].rotation.y += 0.04;   //SQUAME 3
this.snakeGroup.children[0].children[10].children[0].position.x += -0.005;  //PUPILS RIGHT
this.snakeGroup.children[0].children[11].children[0].position.x += -0.005;  //PUPILS LEFT
this.snakeGroup.children[0].children[8].children[11].position.y += -0.01;   //TONGUE
this.snakeGroup.children[0].children[8].children[11].position.z += 0.04;   //TONGUE
this.snakeGroup.children[0].children[8].children[11].rotation.x += -0.02;   //TONGUE
```

Infatti a parte i denti, ogni singolo pezzo, dalla bocca, alle squame superiori, agli occhi esegue un cambio di posizione ad ogni frame. Questo crea un gioco che complessivamente anima la testa in modo ritmico. Non è stato necessario usare nessuna libreria aggiuntiva esterna, perché sono gli oggetti che ad ogni frame vengono spostati.

3.2 La Capra

La capra è stata creata con lo stesso metodo usato per il drago. La particolarità della capra sta nel suo corpo centrale: infatti è stato usato un `blockmesh` per tenere insieme tutte le parti.

Figura 3.2: La Capra.

Nel caso del drago avevamo usato una funzione gruppo. Anche la capra è un modello gerarchico. La gerarchia è molto utile perché ci sono diverse parti che vengono mosse in modo separato. Invece per la testa della capra abbiamo usato un gruppo.

```
build() {
  var blockMesh = new THREE.Mesh(this.blockGeometry, this.blockMaterial);
  blockMesh.castShadow = true;
  blockMesh.receiveShadow = true;
  blockMesh.rotation.set(1.57, 0, 0);
  blockMesh.position.set(0, 0, 0);
  blockMesh.name = "Sheep:Block_0";
```

Agli oggetti che devono essere mangiati dal drago è stato aggiunto un anello circolare che ruota loro attorno. A differenza del drago, nel movimento della capra è stata utilizzata una variabile `t` di tempo, per suddividere il frame in più istanti.

Anche nella capra ogni gruppo di cui è composta esegue un movimento indipendente.

```
update() {
  var t = game.timer.getElapsedTime();
  this.group.position.z += 0.1;
  if (Math.abs(this.group.position.z) >= 25)
```

```
      this.group.position.z = 0.5;
    if (Math.abs(this.group.position.x) >= 20)
      this.group.position.x = 0.5;

    this.group.children[1].rotation.z += Math.sin(3 * t) / 90;
    this.group.children[2].rotation.y += Math.sin(3 * t) / 45;
    this.group.children[3].position.x += Math.sin(3 * t) / 200;
```

3.3 L'uovo e la papera

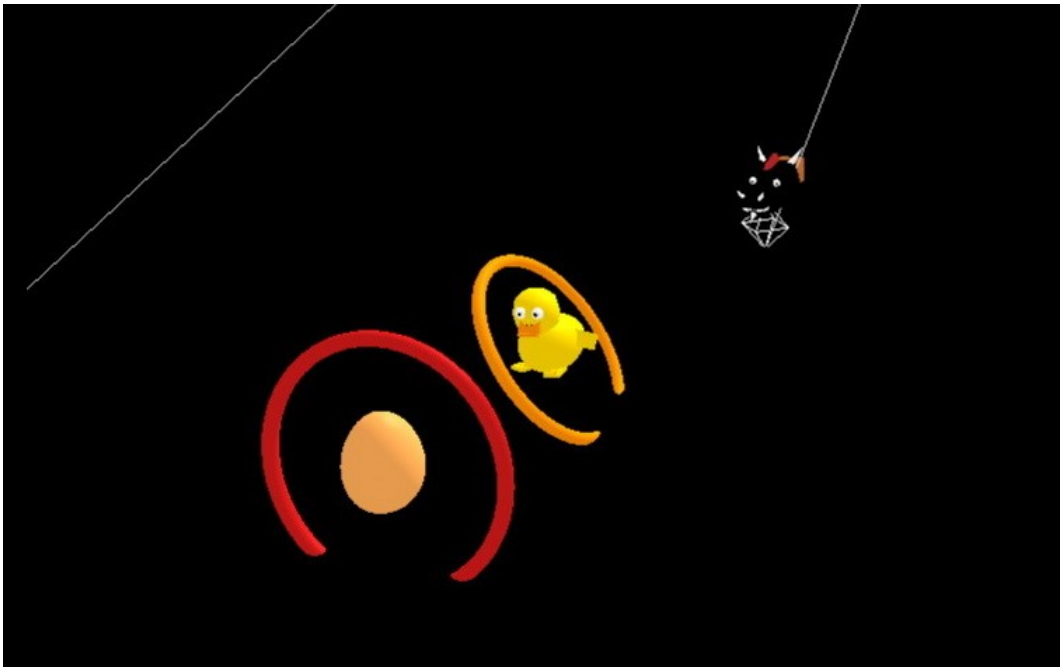

Figura 3.3: Gli oggetti `Egg` e `Duck`.

I modelli utilizzati per l'uovo e la papera sono molto simili, perché la papera è una versione dell'uovo con una serie di aggiunte. Come il drago e la pecora sono dei `blockmesh` basati su un corpo centrale a cui sono agganciati altri oggetti singoli oppure altri gruppi. Ognuno di loro però presenta delle particolarità.

3.3.1 L'uovo

Per quanto riguarda l'uovo, la sua particolarità è nella forma usata per il corpo centrale. Infatti come forma geometrica è stata usata una `THREE.LatheBufferGeometry`, cioè una forma geometrica che prende in ingresso un array (nel nostro caso un `eggarray`) di punti e li unisce per creare appunto la forma geometrica.

```
var eggGeometry = [];
for (var deg = 0; deg <= 180; deg += 6) {
  var rad = Math.PI * deg / 180;
  var point = new THREE.Vector2((0.72 + .08 * Math.cos(rad)) * Math.sin(rad), - Math.cos(rad));
    // the "egg equation"
  eggGeometry.push(point);
}

class Egg {
  this.group = new THREE.Group();
  this.group.name = "EggGroup";
  this.blockGeometry = new THREE.LatheBufferGeometry(eggGeometry, 50);
  this.blockMaterial = new THREE.MeshPhongMaterial({
  color: 0xFFAA55,
  wireframe: false,
  depthTest: true,
});
```

L'uovo a questo punto è già completo. Viene aggiunto un cerchio che gli ruota attorno ed il movimento è solo un cambio di posizione verticale frame dopo frame sull'asse y.

3.3.2 La papera

Il modello gerarchico della papera è basato su un corpo centrale ovoidale (lo stesso creato per l'uovo) e su una serie di altri componenti simili o a gruppi.

Quello che contraddistingue la papera dagli altri modelli è il movimento su un quadrato ideale.

```
causalmove += 1;
if (causalmove < 100) {
  this.group.position.z += 0.05;
  this.group.rotation.y = 3.14;
  this.group.rotation.z = 0;
  }
else if (causalmove < 200) {
  this.group.position.x += 0.05;
  this.group.rotation.y = 4.6;
  this.group.rotation.z = 0;
  }
```

3.4 Paesaggio

Anche per il paesaggio sono stati realizzati diversi modelli, sia per quanto riguarda il suolo, che per il cielo.

Due gli elementi su cui si è focalizzati: due tipi di gruppi di nuvole e tre tipi di piante.

3.4.1 Gruppi di nuvole grandi e piccole

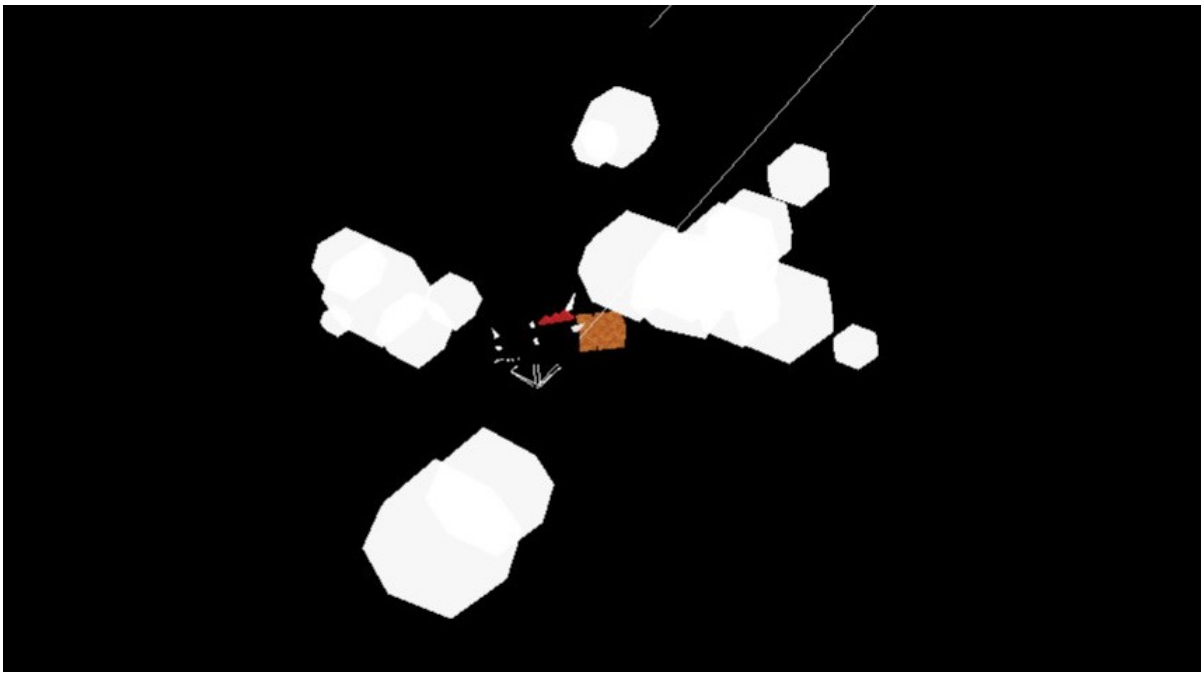

Figura 3.4: Gruppi di nuvole.

Le nuvole sono formate da gruppi geometrici identici fra loro. Ci sono due diversi gruppi di nuvole: un blocco più grande ed uno più piccolo dislocati in maniera sparpagliata nel cielo.

Due le particolarità: in questo caso si è usato un materiale con un grande di trasparenza ed il movimento differenziato delle nuvole in modo che alcune potessero allontanarsi prima ed altre dopo.

```
class Cloud {
  constructor(position) {
    this.position = position;
    this.rotation = new THREE.Vector3(0, 0, 0);
    this.group = new THREE.Group();
    this.group.name = "CloudGroup";

    // this.blockGeometry = new THREE.LatheBufferGeometry(cloudGeometry, 50);
    this.blockGeometry = new THREE.OctahedronBufferGeometry(1, 1);
    this.blockMaterial = new THREE.MeshBasicMaterial({
```

```
    color: 0xffffff,
    wireframe: false,
    depthTest: false,
    transparent: true,
    opacity: 0.70,
    fog: true
    });
  ...
```

3.4.2 Piante

Per il paesaggio sono state create 3 diversi tipi di piante: un albero, una palma ed un cactus a seconda dell'ambiente in cui devono poi essere inseriti.

I modelli sono tutti e 3 gerarchici e tutti e 3 sono stati dotati di movimento.

Figura 3.5: Piante: Albero, palma e cactus.

L'unica diversità fra i vari modelli è il tipo di geometrie utilizzate, i materiali ed i colori.

Capitolo 4

Iterazioni e collisioni

4.1 Iterazioni utente

L'iterazione con l'utente è gestita direttamente tramite il browser e le API di JavaScript. All'interno della classe `Game` viene passato il controllo ad una funzione che filtra tutti gli eventi della tastiera e ritorna una variabile globale letta dalla funzione `update()` della classe `Snake` nella quale si analizzano i codici ASCII dei tasti e si associano, tramite un albero decisionale `switch/case` i movimenti del personaggio.

La funzione `snake.move()` si presenta così:

```
move()
{
  if (!this.isDead)
  {
    switch (globalKeyPressed)
    {
      // [ Z ]
      case (90):
        if (this.isEqual(this.snakeDirection, [0, -1, 0]))
        {
          console.log("Auto eat");
          if (this.length == 0)
            this.isDead = true;
          break;
        }

        this.moveHead(0, +move, 0);
        this.setOrientation(0, +1, 0);
        this.updateBody(0, +1, 0);
        break;
    ...
      // [ A ]
      case (65):
      if (this.isEqual(this.snakeDirection, [-1, 0, 0])) {
      console.log("Auto eat");
      if (this.length == 0) this.isDead = true;

      break;
      }
      this.moveHead(+move, 0, 0);
      this.setOrientation(+1, 0, 0);
      this.updateBody(+1, 0, 0);
      break;
    ...
    default:
    break;
    }
  }
}
```

Il controllo del mouse, associato direttamente al controllo della posizione e rotazione della camera, viene effettuato dalla classe (di THREE.js) `THREE.OrbitControls` all'interno della classe `Game`:

```
  this.controls = new THREE.OrbitControls(this.camera, this.renderer.domElement);
```

4.2 Collisioni

Nell'ambito della libreria `THREE.js` esistono vari metodi efficienti ed efficaci per la gestione e la rivelazione delle collisioni tra oggetti in scena. Questi metodi si basano sulla tecnica del **ray tracing**: tracciare una retta dalla normale della superficie "front" dell'oggetto fino ad una certa distanza, tramite la quale calcolare e quindi enumerare gli oggetti che, tramite prodotto tra matrici di posizione, si intersecano con la retta tracciata dal ray tracing. Se da una parte questo metodo è comodo ed efficace, dall'altra la complessità di implementazione e la notevole richiesta di potenza computazionale ci ha fatto optare per una soluzione più "ingegneristica": meno raffinata ma più efficiente.

Ogni oggetto in scena, in fase di realizzazione (`build()`) genera un ulteriore oggetto non visibile (quindi non renderizzato) di tipo `THREE.Box3()` costruito intorno alla dimensione totale d'ingombro dell'oggetto stesso.

Per esempio nella classe `Snake` questo viene eseguito con:

```
var BB = new THREE.Box3().setFromObject(this.snakeGroup.children[0]);
BB.name = "snakeBB";
```

Si nota come il primo oggetto dell'intera struttura (`group.children[0]`) sia inscritto nel parallelepipedo `BB` (*hitbox*) creato con il metodo `setFromObject()`; La `THREE.Box3` ha una proprietà `name` con il quale si indica il tipo di oggetto associato tramite il quale si effettua il calcolo della collisione.

4.2.1 `Snake.checkCollision()`

Sempre per risparmiare risorse e mantenere un alto frame rate, abbiamo deciso che il calcolo delle collisioni avvenga esclusivamente all'interno dell'oggetto `snake`. Qui la funzione `checkCollision()` crea un array di collisioni (intersezioni tra le varie hitbox presenti in scena).

```
var listCollisions = [];
for (var i = 0; i < game.boxes.length; i++)
{
  var collision = BB.intersectsBox(game.boxes[i]);
  if (collision == true)
  {
    listCollisions.push(game.boxes[i].name);
    if (game.boxes[i].name == "eggBB")
      this.eatEgg();

    if (game.boxes[i].name == "duckBB")
      this.eatDuck();

    if (game.boxes[i].name == "sheepBB")
      this.eatSheep();

    if (game.boxes[i].name == "cloudBB")
      this.length -= 2;
  }
}

game.boxes = []; // Cleaning hit boxes buffer
```

Dove l'array `game.boxes` contiene l'elenco di hitboxes presenti nella scena, il quale ad ogni update del gioco, viene aggiornato.

Capitolo 5

Considerazioni finali

In queste pagine sono state descritte le varie features del progetto proposto che ha visto la sinergia di quattro componenti, ognuno dei quali ha ovviamente curato un singolo aspetto in particolare. La scelta di misurarci in un riadattamento di un gioco arcade cosi famoso ci ha permesso di capire le vere potenzialità di queste piattaforme per campi più affini alla robotica, come la simulazione di processi industriali o modellistica `3D`.