GAME DEVELOPMENT

*(who knows if I’m gonna write something hereeeee)*

27/02/24 (Tiezzi)

Challenges for a sw developer:

\* potentially very complex (parallel, distributed, real time etc.);

\* interdisciplinary competences (graphics, physics, math, etc);

\* specific development methodologies and development frameworks.

*Lol si userà Unity*

Contents:

- Getting started with unity;

- C#;

- Building simple 3D and 2D demos;

- Step-by-step tutorial for building pac-man in Unity;

- Programming aspects of game development;

- Design pattern in game development.

Exam:

- Group project;

- Oral discussion with a presentation (and maybe a report describing design choices,

problems and relative solutions).

Frameworkless game development:

freamworkless approach allows to make optimal use of hw.

Framework-based game development:

Code reuse and a lot of libraries.

29/02/24 (Tiezzi)

From freamworks to game engines

First frameworks were created in-house to support the development of first-party software; later, frameworks became larger game engines, including development tools (renderers, physics engines, collision detection, etc.) integrated in IDE. A game implemented using an engine will get all its features.

Unity characteristics

• it is one of the most popular and used game engines

• it is a powerful and professional-quality tool

• it is, at the same time, approachable for just getting started

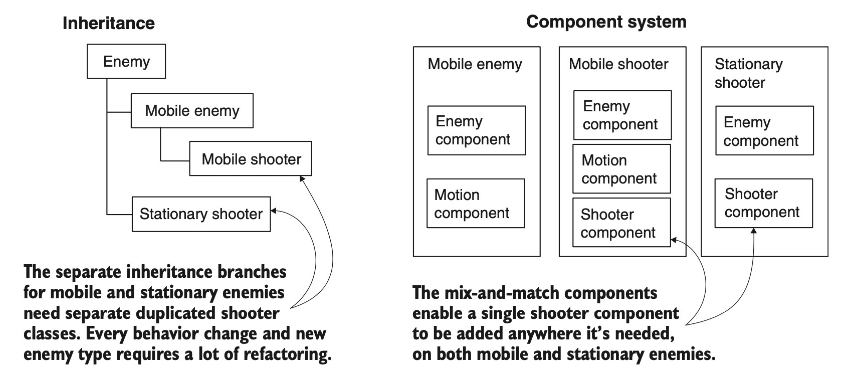
• there are a lot of learning resources (perhaps, too much!)

• it is perfect for our focus on the programming aspect of game development

• it works great for 3D and 2D games (although its original focus was on 3D)

Unity's strengths and advantages

* *Extremely productive visual workflow:* the development workflow is anchored by a sophisticated visual editor; the editor is used to lay out the scenes in your game and to tie together art assets and code into interactive objects, the editor itself can be customized by writing scripts that add new features and menus to the interface.
* *High degree of cross-platform suppor*t: multiplatform in terms of development tools, games can be developed on Microsoft Windows or Apple macOS and multiplatform in terms of deployment targets: games can be deployed to PC, web, mobile, consoles, AR/VR.
* *Modular component system to construct game objects*: components are mix-and-match(combinabili) packets of functionality, and objects are built up as a collection of components.   
  A component system is a different (and usually more flexible) approach to object-oriented programming (based on a strict hierarchy of classes). In a component system, objects exist on a flat hierarchy, and different objects have different collections of components, the component arrangement facilitates rapid prototyping, because you can quickly mix and match components rather than having to refactor the inheritance chain when objects change.



Unity component system is integrated with the visual editor and each component an be attached and detached within the visual editor, of course developer is not limited to building objects only through composition; inheritance in the code can be used, including all the best-practice design

patterns that have emerged based on inheritance.

Unity's disadvantages

* *Combination of visual editor and coding can create difficulties*: because developers can lose track of which objects in the scene have specific components attached.
* *Linking in external code libraries can be difficult*: however, now Unity comes with a Package Manager.
* Unity offers multiple approaches to some functionalities, and it is not always clear which approach you should use.

05/03/24 (Tiezzi)

C# *(Metodi con le maiuscole -\_- )*

Strongly typed language.

To print on screen: Console.WriteLine(string);

To read from the keyboard: Console.ReadLine().

String are Object even though we can use string instead of String, it’s better to use string actually.

String interpolation:

string firstName = “Elia”;

string lastName = “Matteini”;

string name = $”My full name is: {firstName} {lastName}”;

*Long story short: C# è Java ma con i metodi che iniziano con la maiuscola*

07/03/2024 (Tiezzi)

PROCEDURAL PROGRAMMING C#

* C# is typed language;
* We can use:
  + collections;
  + functions;
  + loop, do..while, for ecc…

Remind of past lecture about OOP.

*I can’t believe he’s actually explaining Object Oriented… (dhn) :((*

*Explanation of CONSTRUCTOR method…*

Explanation of GETTER and SETTER methods. They are a bit different from Java.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public class Car {

private String color;

get {color}

set{color = value}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The keyword SEALED blocks the inheritance of a CLASS.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class Animal {

public void animalSound(Console.write(‘Some sound!’);)

}

class Dog : Animal {

public void animalSound() {Console.write(‘Wof woof!’);}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The keyword “:” is used for inheritance.

But if I do

Animal dog = new Dog();

dog.animalSound(); // OUTPUT: Some sound!

In C# there is no dynamic typing, only static typing. Different from Java.

To obtain the same behavior of Java we need to use a special keyword on my Dog class “OVERRIDE”.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class Dog : Animal {

public void override animalSound() {Console.write(‘Wof woof!’);}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Now if I do:

Animal dog = new Dog();

dog.animalSound(); // OUTPUT: Wof woof!

I obtained what I expected.

*ZIO PERAAAAAAAAAAAAAA*

12/03/2024 (Tiezzi)  *porcozzio quanta gente c’è (“lezione noiosa” parole del prof) (macchina tagliaerba is the new elicottero) number one*

*-First lesson in “aretino”*

*Unbelievable oggi spiega le classi astratte (povco)*

The main program will be programmed using the engine, c# will be used to “fill holes” using inheritance or patterns *lmao*.

C# does not support multiple inheritance, but a class can implement multiple interfaces.

Casting is the same of C *(unbelievable)*

*Now he’s explaining Enum but Bettini told us that is bad, so don’t use enum.*

We can work with files using the File class from System.IO

*(Now he’s explaining exceptions ZIO PERA NON SIAMO A PROGRAMMAZIONE)*

Exceptions are objects of class Exception (or child-class)

*(now we are discovering at run time if “super” exist in C#)*

*NON NE POSSO PIU’ ZIO PERA, METTERO’ TUTTI I COMMENTI NOSTRI IN CORSIVO*

The keyword super used in Java in C# is called base

The keyword virtual allows derived classes to override a method.

foreach syntax in C#:

Person[] people = {s1, p, s2}

foreach(Person persona in people) {

persona.surviveToThisLecture(“dhn”);

}

Unity

Unity for Linux, first voice of the web page *(“first voice” sfotti pure, io non l’avevo trovato)*

### “Install the Unity Hub on Linux:”

### “Debian or Ubuntu”

https://docs.unity3d.com/hub/manual/InstallHub.html#install-hub-linux

Scene: the main part of the interface, made up of “components” (objects)

The “scene view” shows what the game world looks like. There are two kind of objects:

* Mesh objects, like physical objects, NPC *and stuff like that…*
* Other objects, like cameras, lights *and shit like that…*

Toolbar: provides buttons for scene navigation and transforming objects

Hierarchy view lists the name of every object and allow to manage and link all the object on the scene.

Inspector panel shows information about the currently selected object and allows you to change some parameters of the object.

Da recuperare la cosa delle scene….

*\*aretino intensifies \**

F, very nice shortcut to “centrate” the view on the main camera, *pay attention to not lose your mind, it’s very hard after the hours before @\_@ U\_\_U ‘X\_\_X*

*Ho appena lanciato unity, credo che il pc stia per scoppiare Rip ahahahah*

—

14/03/2024 (Costa - mancano slide)

19/03/2024 (Tiezzi)

*Today lesson in english lessgooo Ez 😎*

Script component

Game objects are built up as a collection of components.

Script components inherit from the MonoBehaviour class.

MonoBehaviour has two important methods:

Start() -> Called only one time to active the object

Update() -> Called every frame, it draws the frame

You can create a **C# script** form the menu “assets” and a game object from the menu “game object”.

It’s possible to **associate a C# script to a game object** by drag and drop it to the object

It’s possible to use the class Debug to **write to the console**

You can change the default editor of C# by the menu “edit”->”preferences”->”external tools”->”external script editor”

In the 45 minutes left we are gonna build a First-Person-Shooter scene… *(maybe)*

*First-Person-Shooter = How to create walls lmao*

Things to do:

1. Make the walls, external and internal ones
2. Players, and the camera on top of the player
3. Making a script for each “component” like a script for managing the arrows, a script for managing the mouse…
4. Write the movement for the player

Remember the rule of the right hand? We are gonna use the rule of the left hand *zio pera*

Empty Object are used to group object, just to organize the objects; they are not visible

**Lights** 3 different type:

1. point lights: like physical light
2. spot lights: used to hide or (come cazzo si dice evidenziare) come objects
3. *I lost it I’m sorry*

A light has three property: position, intensity and direction

Object have collider to set the boundaries, player have special collider named Character Controller

21/03/2024 (Costa)

GAME ENVIRONMENT

Scene is the main environment of the game, inside it there are some sprites. The sprites are the object on which we can add some properties and make them interact between each other.

We can organize the sprites in layers that share some properties, doing so we can act on a sprites group. Eg: we can stop the collisions between all sprites in a common layer.

Unity defines 5 pre-defined layers.

LAYER CREATION

* On the left panel we can add layer and order it;
* We can define a physics shared between all sprites in the layer:
  + Click on physics.

The sprite and the scene rendering are linked, so pay attention during rendering to not overlap scene objects and sprites! Unity NEEDS to know the corner of our sprites.

SPRITES 8-bit GRAPHICS

* Pixel per unity = 8;
* Filter mode = Point;
* Max size = 32;
* Compression = None.

Without Compression = None the render texture quality is dynamic based on the machine performance.

By default unity blur the corner of the sprites, by doing so the corners are well defined.

MAZE

To define the maze (Eg: like in PacMan) we can place it on a squared grid. We can do it by creating a TILED MAP object.

* Add a tiled map;
  + Rectangular in this case.
* Add it to a layer;
* Rename it MAZE.

Now we can place the wall of the maze inside the tiled map.

Once we have created the tiled map we can draw the wall of maze by using the TILE PALETTE.

How to do that?

* Click on the previous created Titled Map;
* On the bottom right click to open the Tile Palette;
* Create a new folder (Eg: “scene”);
* Inside it create a TILE PALETTE;
* Place all the wall assets inside it and create a PALETTE;
* Start drawing the maze, by selecting an asset and place it in the Tiled Map’s grid.

This part is very boring, so magically the next lesson the maze will be created.

Now we have to avoid our character to walk inside the wall.

PELLETS

On another layer we can start to make pallets (Eg: the pallets that PacMan eats or the coins collectable by Super Mario).

PREFABS  
Game objects that are not only images, are images on which the character can interact and they have a behavior. Pallets (if for example give a super power like mushrooms in Super Mario) can be PREFABS.

Create a PacMan Pellet Prefab:

1. Create new game object and call it PALLET
2. Under inspector -> add component -> Sprite renderer
3. Set property to be the small pellet image
   1. If necessary resize it
4. Under inspector -> Add Component -> Box Collider 2D
   1. The collider adds behavior to a game object and defines what to do when the character collides with it.
   2. Eg: when PacMan collides with a pellet it disappears.
5. Flag property is **Trigger**;
   1. This tells Unity to not take care of what happens when PacMan collides on the pallet. Is the developer that defines the behavior.
6. Resize the hit box if necessary.

Now we have to associate our prefabs to a grid Titled Map’s cell.

We want to place the pallets in the middle of the wall’s maze, so we need to create a RULE TILE.

RULE TILE: cell that follows some rules defined by the developer.

* Right click inside folder Tiles
* Create -> 2D -> tiles -> Rule Tile
* Give it a name (eg. PalletTile)
* Set the pallet prefabs

FINISHHHHHH.

26/03/2024 (Tiezzi) less people in da room then the other lessons

Today we will create a script that allows the player to move and the camera as well.

The script to move the player are C# script and they have to be positioned in the “Assets”

First thing first pull unity’s classes into the script

using UnityEngine;

To rotate the player we can use the method Rotate of the class transform

transform.Rotate(0, speed, 0, Space.Self); *//(x, y, z, relativeTo)*

This command makes the player rotate on the y coordinate at a certain speed.

When the script is created you have to attach it to the player easily through drag-and-drop and to see if it’s really attached check if the script name is listed on the Inspector panel.

Once the script is saved you can change the variables of the script in the Inspector panel, but this is possible only if the variables are public.

Now let’s see how to implement the MouseLook, in this way the player vision will be consistent with the mouse movement.

*Tiezzi doesn’t like comments on the code, the code has to speak by it self (Bettini school hereeee)*

*ha fatto copia incolla dalle slide che non ha pubblicato non so riuscito a copiare il codice e mi so perso roba da riprendere con le slide sempre per il mouse e euler angles*

*Su lilik ho caricato il libro da cui il Tiezzi prende gli script*

Keyboard input to move the player

transform.Translate(0, speed, 0);

this command allows to move the player

Credo che prima manchi la lezione del costa

09/04/2024 (Tiezzi) in italiano less goooo “Towards DOOM”

To move the player we can use the method Translate from transform class.

To guarantee that the games will perform the same way in each computer we have to multiply the speed velocity with the frame rate of the machine. To realize such things we can use Time.deltaTime

transform.Translate(deltaX \* Time.deltaTime, 0, deltaZ \* Time.deltaTime);

Example code:

public class FPSInput : MonoBehaviour

{

public float speed = 6.0f;

*// Update is called once per frame*

void Update()

{

float deltaX = Input.GetAxis("Horizontal") \* speed;

float deltaZ = Input.GetAxis("Vertical") \* speed;

transform.Translate(deltaX \* Time.deltaTime, 0, deltaZ \* Time.deltaTime);

}

}

Now the player is able to go through walls *(è fortissimoooo)* , to fix this we need to:

* retrieve the character controller -> return a CharacterController object
* use the method Move of the class CharacterController, this method take a vector of the direction. Fanculo lo dico in italiano, per calcolare il vettore risultante la soluzione è fare la somma vettorieale, ma la somma vettoriale determina anche la sua intensità che quindi sarebbe maggior di quella che volevamo, per questo ci viene in aiuto il metodo Vector3.ClampMagnitude(movement, speed) che modula le velocità.  
  To transform the movement from local to global coordinates we can use the method transform.TransformDirection(movement)

Example code:

public class FPSInput : MonoBehaviour

{

public float speed = 6.0f;

private CharacterController charController;

void Start() {

charController = GetComponent<CharacterController>();

}

*// Update is called once per frame*

void Update()

{

float deltaX = Input.GetAxis("Horizontal") \* speed;

float deltaZ = Input.GetAxis("Vertical") \* speed;

*//transform.Translate(deltaX \* Time.deltaTime, 0, deltaZ \* Time.deltaTime); //OLD*

Vector3 movement = new Vector3(deltaX, 0, deltaZ);

movement = Vector3.ClampMagnitude(movement, speed);

movement \*= Time.deltaTime;

movement = transform.TransformDirection(movement);

charController.Move(movement);

}

}

Right now the player can fly, we don’t wanna fly so we can add movement.y = gravity; defining the gravity variables like the earth gravity public float gravity = -9.8f;

Make things easily:

* We can use the RequireComponent attribute to ensure that other components needed by the script are also attached.
* Add the AddComponentMenu attribute to the top of the script to add it to the component menu in Unity’s editor; then, the script can be selected when we click Add Component at the bottom of the Inspector.

[RequireComponent(typeof(CharacterController))]

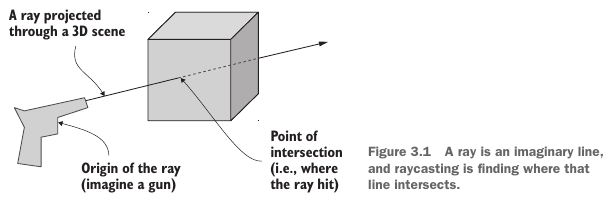
[AddComponentMenu("Control Script/FPS Input")]

public class FPSInput : MonoBehaviour

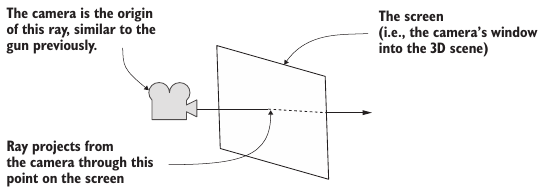
...

SHOOT

To replicate DOOM now we need to shoot from a shotgun. In order to shoot we can use the raycasting and as the name suggests is a cast of a ray into a scene; once that we have a ray that goes straight through our screen we just need to check the collision of that ray.



We’ll implement shooting by projecting a ray that starts at the camera and extends forward through the center of the view. Unity provides the ScreenPointToRay() method to perform this action. The method creates a ray that starts at the camera and projects at an angle, passing through the given screen coordinates. Usually, the coordinates of the mouse position are used for mouse picking (selecting the object under the mouse), but for first-person shooting, the center of the screen is used. Once you have a ray, it can be passed to the Physics.Raycast() method to perform raycasting using that ray.



Clearly the RayShooting script will be attached to the camera.

Example code:

public class RayShooter : MonoBehaviour

{

private Camera cam;

void Start() {

cam = GetComponent<Camera>();

}

void Update() {

if (Input.GetMouseButtonDown(0)) {

Vector3 point = new Vector3(cam.pixelWidth/2, cam.pixelHeight/2, 0);

Ray ray = cam.ScreenPointToRay(point);

RaycastHit hit;

if (Physics.Raycast(ray, out hit)) {

Debug.Log("Hit " + hit.point);

}

}

}

}

out is the keyword to force pass by reference value as parameter.

16/04/2024 (Costa) *aiuto mai fatto lezione con il Costa, cazzo ci faccio qui*

Pac man uses a buffer to store the next direction if at a certain point the user gives an input direction that is not allowed. The stored direction will be executed when the pac man finds a corridor in that direction. To do this we can use a Raycast and shoot a ray a box from the center of pac man to find a free corridor; we’ll use a ray like a box of the dimension of pac man because it’ll change direction when is exactly at the center of the corridor otherwise it’ll be stuck on the wall.

Here we need to pay attention to set the layer of the element that the ray can collide with because we want to detect the collision just with the wall's maze.

Clearly the buffer where we store the direction has dimension 1.

*Ma che cazzo sono le sprite, non fa altro che dire sprite.*

The right and left endless corridor actually are connected, so if pac man goes through th corridor ‘til the end it will despawn and respawn on the corridor at the other side. To implement this function we have to position a warp (empty box with dimension the same as the corridor) at the end of the two corridors and then add a script to make the magic that changes the position of pac man.

EATING PELLETS

During the game pac man has to eat pellets, in order to make him(it?) able to do this pac man and the pellets need to interact with each other.

There are two types of pellets in the game, the small one and the big one. Each of the two type has different result, clearly we can take advantages by the fact that C# is an object oriented language and uses the properties of the hereditary.

**18/04/2024 Costa**

è più ragionevole far capire alla pallina che è stata mangiata piuttosto che far capire a pacman che ha mangiato una pallina.

Ogni elemento del gioco deve avere il suo script con le sue proprietà.

Creiamo adesso i GHOST.

**GHOST**

Sono dei prefab, ai quali bisogna aggiungere uno script movement come a pacman. In questo caso però il movimento è autonomo e non stimolato dall’utente.

I fantasmi hanno delle sprites particolari, perché gli occhi cambiano a seconda della direzione in cui si stanno muovendo.

Inoltre hanno un comportamento perché devono inseguire pacman e fuggire da lui durante il powerup.

Posso aggiungere una maschera di colore alle sprite per creare sprite di colore diverso a partire da una sola.

**CPU and AI**

La strategia di gioco può essere ottima (caso in cui matematicamente parlando è possibile codificare delle regole affinché il computer possa sempre vincere) oppure possiamo settare noi la difficoltà in modo che la sfida sia alla pari con l’utente.

In pacman esiste una strategia definita dall’autore. I giochi moderni con tecniche di machine learning possono apprendere nuove strategie giocando.

Nel nostro caso ogni fantasmino ha una strategia diversa, per fare ciò usiamo un modello di MACCHINA A STATI FINITI.

Posso cambiare la mia strategia (es. passo da strategia di attacco a difensiva e viceversa) al verificarsi di un determinato evento (es. powerup di pacman).

FINE

**30/04/2024 (Tiezzi) *Last Tiezzi’s lessonnnn***

Code to add the indicator for aiming

void OnGUI() {

size of this font.

int size = 12;

float posX = cam.pixelWidth/2 - size/4;

float posY = cam.pixelHeight/2 - size/2;

GUI.Label(new Rect(posX, posY, size, size), "\*");

}

(Questo cursore fa caha, nelle slide del Tiezzi dovrebbe esserci la modifica per avere una croce rossa☭ come mirino)

To set the cursor invisible

void Start() {

cam = GetComponent<Camera>();

Cursor.lockState = CursorLockMode.Locked; // set cursor invisible

Cursor.visible = false;

}

The Update() method is executed entirely each frame, if there is something that we need to split on several frames we can use the coroutine, which is not related to concurrency but allows us to execute the code inside the Update() method without starting over every frame.

To know that the enemy has been hit by the projectile alla we have to do is

if (Physics.Raycast(ray, out hit)) {

*// Retrieve the object the ray hit.*

GameObject hitObject = hit.transform.gameObject;

ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();

*// Check for the ReactiveTarget component on the object.*

if (target != null) {

target.ReactToHit();

} else {

StartCoroutine(SphereIndicator(hit.point));

}

}

So, we retrieve the object that the ray hit (RaycastHit hit object) and if the object hitted is an enemy we can call the method to kill it otherwise, if the object is a wall, do something else like shoot a ball that disappears after 1 second.

*WanderingAI da recuperare*

**PREFAB**

Prefabs are game objects that are not part of the scene.

—----------------------------------------------------------------------------------------------------------------

PROGETTO

Idea: Simil risiko Players vs Com, Players vs Players

<http://risiko.it/wp-content/uploads/2017/10/Regolamento-Risiko.pdf>

**Grafica:**

* **Mappa:** la mappa è in 2D, presa da qui: <https://pixabay.com/vectors/board-of-risk-risk-game-of-risk-4308773/>
* **Stati:** ogni stato è cliccabile, al click il giocatore può vedere le informazioni dello stato:
  + Giocatore proprietario dello stato;
  + Numero di armate presenti;
  + Nome dello stato;
  + Continente di appartenenza

**IA: boh chi cazzo la sa fare**

**Online:**

* **Creazione lobby:** Il giocatore “host” può creare una stanza di gioco alla quale sarà assegnato un codice identificativo univoco, gli altri giocatori “client” si potranno collegare alla lobby inserendo il codice della lobby.
* **Macchina a stati:** il server è il gestore delle lobby, i client sono i “visualizzatori” del gioco, quindi la grafica viene elaborata dai giocatori, il server tiene conto soltanto dello stato e della logica del gioco.
* **Linguaggio:** per semplicità di implementazione il server è scritto in Python, DA VALUTARE se scrivere delle api per utilizzare il server da parte del client o mandare dei JSON dal server ai client

**Gameplay:**

**SETUP INIZIALE**

* **Modalità di gioco:**
  + Tradizionale: la partita proseguirà fino a quando uno dei giocatori non avrà raggiunto l’obiettivo indicato sulla carta “obiettivo segreto”.
  + TIME ATTACK: decidere un tempo di gioco al termine del quale si concluderà il giro, se uno dei giocatori ha raggiunto l’obiettivo segreto questo sarà il vincitore, altrimenti, la vittoria andrà al giocatore che controllerà il maggior numero di Punti Vittoria, sommando quelli dei territori sotto il suo controllo. In caso di parità, vincerà il giocatore con il maggior numero di armate in campo. In caso di ulteriore parità la partita termina con un armistizio.
* **Ordine dei giocatori**: il server fa lanciare un dado a ciascun giocatore e da esso decide l’ordine di gioco;

STRUTTURA MESSAGGI:

GAME\_ORDER: idPlayer-position, idPlayer-position (S → C) Il server manda l’ordine di gioco a tutti i giocatori

EXTRACTED\_NUMBER: number (S → C) Il server manda il numero estratto al giocatore

GAME\_ORDER\_EXTRACTED\_NUMBERS: idPlayer-extracted\_number, idPlayer-extracted\_number (S → C) Il server comunica ai client i numeri estratti da tutti i giocatori

* **Scelta dell’armata:** nell’ordine prestabilito i giocatori scelgono la propria armata

IS\_YOUR\_TURN: TRUE/FALSE -> (S -> C) gestione dei turni in generale

AVAILABLE\_COLORS: color1, color2 -> (S -> C) LISTA DEI COLORI DISPONIBILI

CHOSEN\_ARMY\_COLOR: id-color -> (C -> S) Comunica il colore scelto

* **Distribuzione dell’armata:** ogni giocatore dispone di una dotazione iniziale così composta:
  + 35 armate se si gioca in 3
  + 30 armate se si gioca in 4
  + 25 armate se si gioca in 5
  + 20 armate se si gioca in 6

INITIAL\_ARMY\_NUMBER: number -> (S -> C) Comunica il numero di armate iniziali

* **Distribuzione carte obiettivo:** ogni giocatore riceve una carta obiettivo (segreta), sarà il suo obiettivo di gioco

OBJECTIVE\_CARD\_ASSIGNED: json(Card) -> (S -> C) Comunica l’obiettivo che gli è stato assegnato (tramite json)

* **Distribuzione territori:** Secondo l’ordine di gioco il server distribuisce una carta territorio a testa fino all’esaurimento delle carte;

TERRITORIES\_CARDS\_ASSIGNED: Card1, Card2, Card3 -> (S -> C) Comunica la lista di territori che gli è stata assegnata

* **Assegnazione delle armate nei territori**:
  + il server distribuisce automaticamente un’armata per ogni territorio ricevuto;
  + le armate extra vengono distribuite dai giocatori 3 armate per turno, fino all’esaurimento delle armate.

Al proprio turno:

NUMBER\_OF\_ARMY\_TO\_ASSIGN\_IN\_THIS\_TURN: number -> (S -> C) Comunica il numero di tank da assegnare in questo turno

UPDATE\_TERRITORIES\_STATE: Card1, Card2, Card3 -> (C -> S) Appena passato il turno comunica il nuovo stato dei territori, controlli già fatti lato client

* **Ritiro delle carte territorio:** il server ritira le carte territorio dai giocatori e le tiene a disposizione durante il gioco

**FASE DI GIOCO**

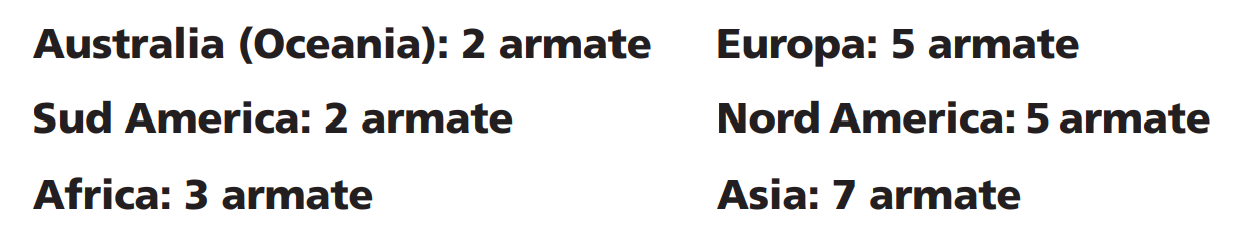
* **Fase 1 “Rinforzo”:** 
  + il server distribuisce tante armate quanti sono i suoi territori diviso 3 (arrotondato per difetto), il giocatore distribuisce sul momento le armate ricevute in questa fase di gioco

Per il turno di ogni player:

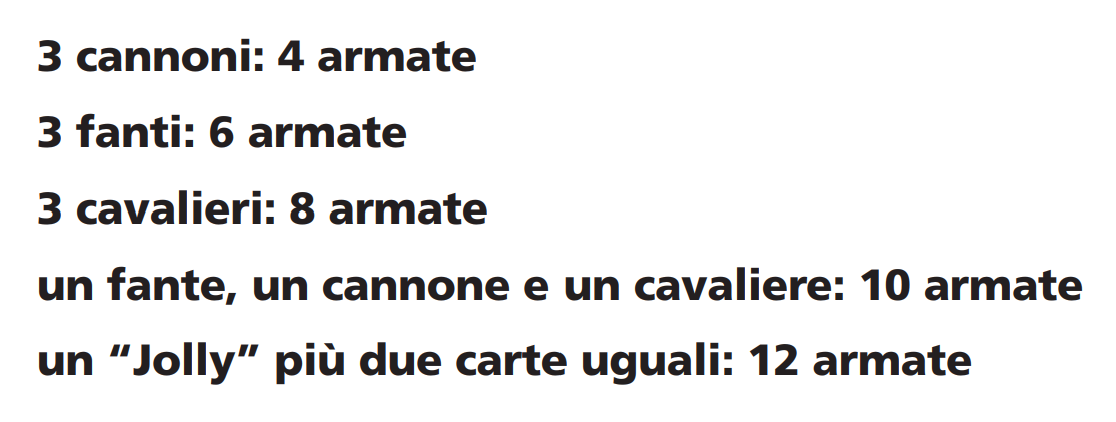
ADDITIONAL\_ARMY\_TO\_ASSIGN: number -> (S -> C) Comunica il numero di armate bonus che ottiene all’inizio del turno, comprese le armate derivanti dall’occupazioni di interi continenti (punto sotto)

UPDATE\_TERRITORIES\_STATE: Card1, Card2, Card3 (vedi sopra)

* + se il giocatore occupa un intero continente riceve armate extra in funzione del continente:



* + il giocatore può giocare un tris di carte per ricevere armate extra così combinate: (IN SOSPESO)



* **Fase 2 “Combattimento”:**
  + Lo scopo di un attacco è l’eliminazione delle armate nemiche dai territori confinanti per consentire alle proprie armate l’occupazione dei territori avversari.
  + Il giocatore non è obbligato ad attaccare ad ogni turno di gioco, ha però l’obbligo di lasciare sempre almeno un’armata di presidio su ogni territorio in suo possesso. In altre parole, non può attaccare partendo da un territorio presidiato da una sola armata.
  + L’attacco può essere effettuato soltanto verso uno stato confinante;
  + Il giocatore sceglie lo stato confinante da attaccare e decide (massimo 3) con quante armate attaccare;
    - **Attaccante:** sceglie con quante armate attaccare;
    - **Difensore:** sceglie con quante armate difendere;
    - **Attaccante/Difensore:** lanciano i dadi e si confrontano i risultati
      * Il dado con il punteggio più alto ottenuto dall’attaccante si confronta con il punteggio più alto del difensore;
      * Se è maggiore il punteggio dell’attaccante, il difensore dovrà togliere dal territorio attaccato una delle sue armate (riponendola fra quelle in dotazione). In caso contrario, sarà l’attaccante a dover ritirare una delle sue armate dal territorio dal quale ha sferrato l’attacco. In caso di pareggio vince sempre il difensore.
      * Se entrambi hanno lanciato più di un dado si confronta il secondo punteggio più alto dell’attaccante con il secondo punteggio più alto del difensore seguendo la stessa procedura.
    - **Attaccante:** può continuare il combattimento e rinnovare la sfida all’avversario per tutte le volte che vuole, sino alla distruzione delle armate avversarie situate sul territorio di suo interesse, oppure decidere di sospendere l’attacco. Il giocatore di turno può sferrare tutti gli attacchi che vuole, da un solo territorio o anche da altri in suo possesso, fino a che è in grado di conservare sui territori da cui vuol fare partire l’attacco almeno due armate.
      * Quando l’attaccante distrugge tutte le armate su un territorio nemico, deve occuparlo con le armate che hanno partecipato all’ultima battaglia.
      * Se lo desidera, però, può trasportare, partendo dal territorio da cui ha sferrato l’attacco, tutte le armate che vuole. Inoltre, può utilizzare il territorio conquistato come base per un ulteriore attacco.
      * Alla fine del proprio turno, ogni giocatore, se ha conquistato uno o più territorio, pesca una (una sola) carta dal mazzo.
* **Fase 3 “Spostamento strategico”:**
  + Alla fine del turno il giocatore può scegliere, a prescindere che abbia attaccato o meno, se spostare una parte delle sue armate, è consentito un solo spostamento;

UPDATE\_TERRITORIES\_STATE: Card1, Card2 (vedi sopra)

TURN\_OVER: idPlayer -> (C -> S) Comunica che, a seguito dello spostamento strategico, il client ha finito il turno

* + Nel momento in cui il giocatore decide di compiere lo spostamento strategico non può più effettuare attacchi e il suo turno si considera concluso.
* **Fase finale:**
  + **Eliminazione di un giocatore:** Quando l’attaccante distrugge l’ultima armata dell’ultimo territorio di un avversario, si impadronisce anche delle sue carte. Il perdente è allora definitivamente eliminato dalla partita;
  + **Gli obiettivi segreti:** Nel momento in cui un giocatore ha raggiunto l’obiettivo segreto indicato sulla carta in suo possesso, la partita si conclude con la sua vittoria