Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Generación de texturas para diferentes tipos de videojuegos en *Unity* con *Stable Diffusion*

Grado en Ingeniería Informática

Generating textures for different videogame types in Unity using Stable Diffusion

José Nicolás Cabrera Domínguez

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **José Demetrio Piñero Vera**, con N.I.F. 43.774.048-B profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Generación de texturas para diferentes tipos de videojuegos en Unity con Stable Diffusion"

ha sido realizada bajo su dirección por **José Nicolás Cabrera Domínguez**,

con N.I.F. 42.237.465-N.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 13 de julio de 2023

La Laguna, 13 de julio de 2023

Agradecimientos

Mi más sincera gratitud va para mi familia por su apoyo incondicional, tanto emocional como financiero. Su cariño, sacrificio y paciencia han sido elementales para mi triunfo a nivel académico.

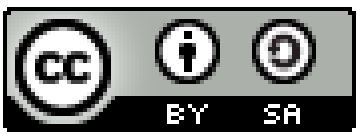
También, quiero agradecer a mis amigos por convertir este viaje en algo más disfrutable. Las relaciones que he tenido por el camino no tienen precio, y me han provisto de una motivación y apoyo emocional importantísimos.

A Expero Inc., la empresa en la que trabajo, por dejarme todo el tiempo que necesitaba para finalizar mis estudios y apoyarme a la hora de realizar este trabajo con tutorización y consejo.

Gracias a mis profesores y al equipo de la Universidad por su dedicación y conocimiento. Por compartirlo y por guiarme en estos años.

Finalmente, agradecer a mis tutores por la guía y el apoyo en el desarrollo del proyecto, por transmitirme su ilusión y apoyarme con recursos y ánimos por el camino.

Licencia



Resumen

Con el rápido avance de los modelos generativos de inteligencia artificial, han tenido lugar multitud de avances en cuanto a la generación de gráficos y el arte. Estos modelos de Inteligencia Artificial han demostrado ser muy potentes a la hora de generar imágenes de alta calidad, ofreciendo una nueva vía para mejorar la fidelidad visual del contenido digital. En este trabajo, se pretende explorar la integración de dichos modelos con Unity, una de las herramientas de desarrollo de videojuegos más famosas, con el objetivo de crear una herramienta para desarrolladores que ayude al texturizado de objetos.

La habilidad de iterar rápidamente entre texturas durante el desarrollo de un videojuego puede ser crucial para poder conseguir experiencias inmersivas y creativas. Los métodos tradicionales para generar texturas suelen venir de la mano con procesos que requieren de mucho tiempo. Esto puede limitar la capacidad de exploración creativa para los desarrolladores. Al aprovechar la potencia de los modelos generativos de Inteligencia Artificial, los desarrolladores pueden automatizar gran parte del proceso, permitiendo así el prototipado rápido de ideas y diseños visuales.

La integración de modelos generativos de Inteligencia Artificial en Unity presenta varios desafíos técnicos. En este trabajo, se investigan estos desafíos y se les propone una solución que aprovecha toda la agilidad de estos modelos, integrándolos a su vez de forma fluida en el entorno de Unity.

Palabras clave: IA, Inteligencia Artificial, Unity, Generación de Texturas, Desarrollo de Videojuegos, Stable Diffusion

Abstract

With the fast advance we're seeing on Artificial Intelligence generative models, there has been a multitude of advances on graphics generation and art. These Artificial Intelligence models have demonstrated their power when generating high quality images, offering a new way of improving the visual fidelity of digital content. In this project, we aim to explore the integration of said models with Unity, one of the most famous video game design tools, with the objective of creating a developer tool that assists on texturing objects.

The ability to quickly iterate between textures during a game's development can be crucial in order to obtain immersive and creative experiences. Traditional texture generating methods can bring time-costly processes. This can limit the creative exploration capacity for developers. When harnessing the power of these generative AI models, developers can automate a big part of this process, enabling themselves to quickly prototype ideas and visual designs.

The integration of AI generative models in Unity presents various technical challenges. In this project, those challenges are investigated and solutions are proposed that leverage the agility of these models, integrating them seamlessly in the Unity environment.

Keywords: AI, Artificial Intelligence, Unity, Texture Generation, Video Game Development, Stable Diffusion

Índice general

Índice de figuras

Capítulo 1 Introducción

Los modelos de Inteligencia Artificial han revolucionado el campo de la computación permitiendo la generación de imágenes realistas y de gran calidad. Estos modelos, que suelen basarse en técnicas de *deep learning*, son capaces de aprender patrones y de *datasets* existentes y de generar contenido nuevo basado en ese conocimiento adquirido.

En el mundo de la generación de imágenes, los modelos de IA generativos pueden producir imágenes muy diversas y sorprendentes. A base de entrenarlos en *datasets* de imágenes, estos modelos pueden aprender a capturar patrones y características de los datos de entrenamiento. Después, pueden generar imágenes que repiten estilos, texturas, o características con las del *dataset* original. Esta capacidad se está aprovechando para crear arte, simular escenas reales, e incluso generar imágenes de personas o lugares que no existen.

En el campo del diseño y desarrollo de videojuegos, los modelos de generación de IA están empezando a ganar atención. Se pueden integrar en los flujos de trabajo del desarrollo para ayudar a los diseñadores a crear mundos virtuales. Incluso, se pueden entrenar los modelos en conjuntos de *assets* que ya están siendo utilizados, para que así generen cosas con sentido dentro de un marco preestablecido. También se pueden utilizar estos modelos para generar contenido de forma procedural, ofreciendo así variaciones y contenido único para el jugador.

La integración de modelos generativos de *IA* en el diseño de videojuegos no solo agiliza el proceso de creación sino que también amplía los límites de la creatividad. Con la mejora y el avance de estas técnicas, podemos esperar cada vez más mejoras en los mundos que nos encontramos.

1.1 Objetivos

En un trabajo experimental como es este, el asentar objetivos concretos puede resultar complicado. Es por esto que ha tenido lugar un estudio previo del estado del arte, en búsqueda de una idea concreta a la que aplicar las técnicas. Al acabar todo el proceso, y gracias a los tutores que ayudan a definir el alcance del proyecto, se definen

de la siguiente manera los objetivos:

El producto final del trabajo se trata de una herramienta de desarrollador para el motor de videojuegos Unity. Esta herramienta proporciona al diseñador del videojuego la capacidad de texturizar de forma rápida los objetos de la escena, utilizando como texturas las generaciones de un modelo de Inteligencia Artificial: Stable Diffusion.

Además, se proporciona una funcionalidad de versionado de las texturas. Es decir, las generaciones que se van aplicando como texturas al objeto se van guardando, y se puede elegir cuál aplicar de entre todas las anteriores, para poder comparar iteraciones de forma cómoda durante el desarrollo.

1.2 Estado del Arte

Antes de empezar con el desarrollo, ha tenido lugar una investigación del panorama en cuanto a las aplicaciones de modelos de inteligencia artificial generativos en videojuegos.

Ya que al principio esta investigación trataba de tomar inspiración para establecer los objetivos concretos del trabajo, esta búsqueda fue en un principio bastante amplia.

Existen proyectos muy curiosos y muy visuales, como por ejemplo algunos que importan sistemas NeRF (Neural Radiance Fields) en Unity [1].

Tras investigar y consumir diversos recursos, la idea que se presenta más atractiva para el desarrollo del trabajo es la de integrar de alguna forma modelos generativos de imágenes en el desarrollo de videojuegos. Para esto se consultan varios proyectos para, primero, ejecutar este tipo de modelos generativos de forma sencilla, y segundo, conectarlos con los videojuegos desarrollados de alguna forma.

Los dos proyectos que se utilizan más como recurso para este trabajo, y que finalmente se utilizan como influencia e inspiración directa, están relacionados estrechamente. Se trata, por una parte, de la *stable-diffusion-webui* del autor *Automatic1111* en GitHub [2]. Este proyecto proporciona la capacidad de ejecutar de forma local, consumiendo los recursos del ordenador donde se ejecuta, un modelo de Stable Diffusion [3].

Principalmente, la interacción con el modelo se hace a través de una interfaz web [Figura 1.1] donde tocar parámetros y cambiar opciones para las generaciones.

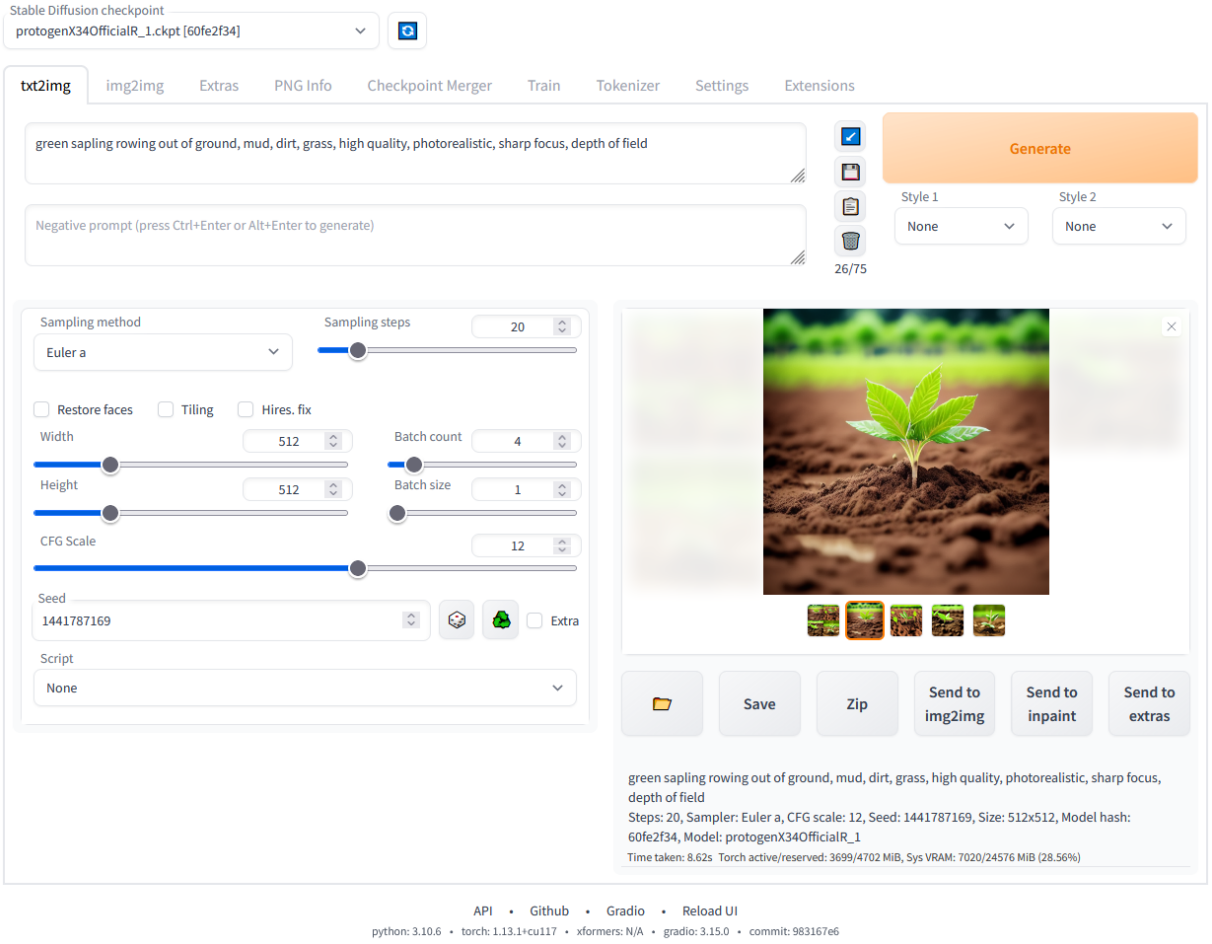


Figura 1.1: Interfaz Web de *Stable Diffusion WebUI*

Además de esta forma de interactuar con el modelo, se proporciona otra distinta: una API [4]. En esta API, podemos hacer peticiones pasando como datos de entrada todos los parámetros que están presentes en la UI. Como respuesta a las peticiones, recibiremos la imagen de resultado codificada.

Por otra parte, el trabajo realizado se apoya en gran medida en los conectores para Unity que ha desarrollado un usuario llamado *JPhilipp* [5]. Con estos conectores para Unity se simplifican mucho las llamadas a esta API. Muchos valores, al no ser especificados, se asumen por defecto y se hacen las llamadas y la decodificación de la imagen resultado, haciendo así mucho más sencillo el utilizar la API de *Automatic1111*.

Capítulo 2 Metodología

En este capítulo, veremos cómo hemos hecho la implementación de la herramienta para cumplir los objetivos establecidos. Se presenta un esquema inicial [Figura 2.1] que conecta todas las partes del desarrollo. Para cada parte funcional de la herramienta, vemos en qué apartado se desglosa dentro de este capítulo. Se distinguen tres componentes principales: la funcionalidad que interconecta el modelo generativo con Unity, la funcionalidad de versionado de texturas y/o escenas, y finalmente la interfaz customizada que permite acceder a estas funcionalidades desde el propio editor.

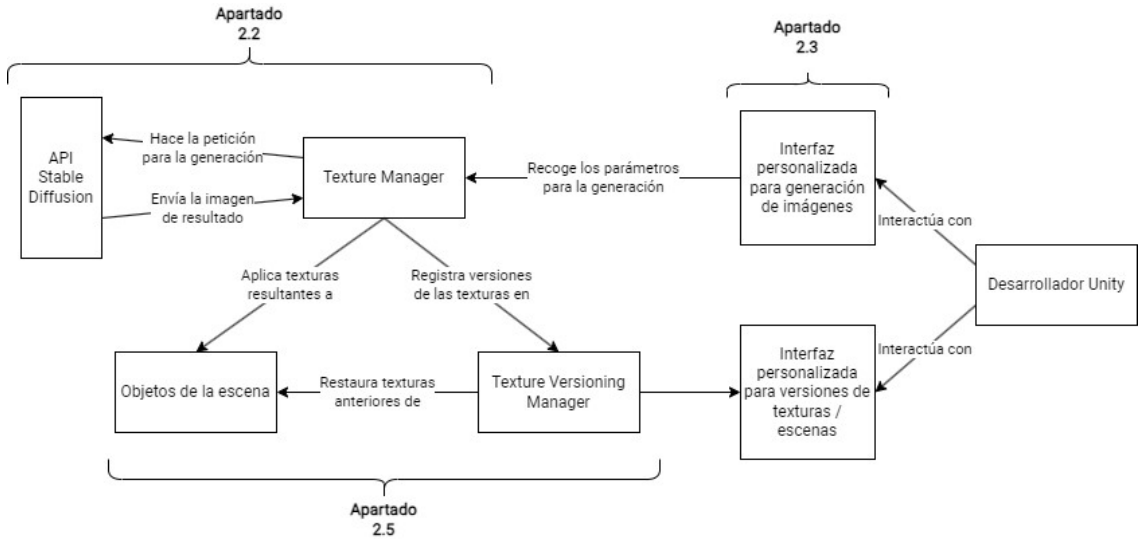


Figura 2.1: Diagrama de funcionalidades de la aplicación con correspondencia a apartados de la memoria

2.1 Tecnologías Utilizadas

2.1.1 Unity

Unity [6] es un motor de desarrollo de videojuegos muy versátil y utilizado que permite a los desarrolladores crear videojuegos en 2D y 3D para una variedad de plataformas. Provee una interfaz y un conjunto de herramientas comprensivo, y en su mayoría se apoya en el lenguaje de programación C#.

El motor ha ganado mucha popularidad por su flexibilidad, amplia compatibilidad con plataformas y su gran comunidad que comparte recursos y soluciones a través de los foros. Es una herramienta a tener en cuenta tanto para desarrolladores independientes como para equipos más grandes.

Unity se ha convertido en una herramienta importante no solo para el desarrollo de videojuegos. Su alcance llega a otras industrias a la hora de crear experiencias interactivas y simulaciones. Se utiliza en campos como arquitectura y diseño [7], la industria automotriz y aeronáutica [8], e incluso el mundo de la animación y la filmografía [9].

2.1.2 Visual Studio Code

Visual Studio Code [10] es un editor de código fuente ligero y muy extensible desarrollado por Microsoft. Es ampliamente popular entre desarrolladores y ofrece muchas capacidades que lo hacen muy útil para equipos de cualquier tamaño y tipo. Visual Studio Code provee un entorno de desarrollo rico y personalizable con soporte para infinidad de lenguajes de programación.

Ofrece cosas muy útiles como integración del sistema Git, y un ecosistema que permite mejorar la experiencia de desarrollo con opciones como el debugging, linting, formateo de código, etc.

Su flexibilidad y facilidad de uso lo hacen una elección excelente para desarrolladores individuales.

2.1.3 Visual Studio

Visual Studio [10] es una alternativa a la herramienta anterior. Pasa a ser un IDE (entorno de desarrollo integrado) completo, y como tal ofrece muchas más funcionalidades.

Se suele utilizar mucho al trabajar con Unity ya que Visual Studio fue creado para trabajar con el lenguaje de programación C#, y por ello tiene muchas utilidades que

favorecen y agilizan el trabajo.

Durante el desarrollo se ha utilizado junto con Visual Studio Code para tareas concretas que necesitaban de esta sinergia añadida con C#.

2.1.4 Git + Github

Git [11] es un sistema de versionado distribuido que permite a los desarrolladores gestionar y hacer el seguimiento de los cambios al código de forma eficiente. Permite a los usuarios crear ramas, hacer cambios, fusionar cambios y gestionar conflictos entre versiones. Además, se mantiene un historial completo de modificaciones, haciendo más fácil el revertir a versiones anteriores.

Por otra parte, GitHub [12] es un sistema de hosting basado en web para repositorios Git. Con GitHub, los desarrolladores pueden compartir el código con otros, crear y revisar peticiones de cambios, tener un control sobre posibles fallos y hacer un seguimiento del progreso del proyecto.

Usar Git y Github en sintonía para un proyecto ofrece numerosas ventajas. Esto provee un repositorio centralizado y accesible para el almacenaje y versionado del código. Además, tener el repositorio en Github provee de una cara pública donde otros usuarios pueden acceder a ver el trabajo, aprender y sugerir cambios y compartir ideas.

2.1.5 Trello

Trello [13] es una herramienta visual de gestión de proyectos, seguimiento de tareas y flujos de trabajo. Se definen tareas, representadas como “tarjetas” en un tablero. Estas tarjetas se pueden colocar en secciones del tablero que representan que la tarea está en proceso, iniciada, terminada, etc. Además, se pueden personalizar con etiquetas, fechas, anotaciones y comentarios.

Durante el desarrollo se ha utilizado la herramienta para subdividir los objetivos en tareas y hacer un seguimiento de las mismas para mayor organización.

2.1.6 Stable Diffusion

El concepto de difusión, en la física, se trata de un proceso que consiste en el desplazamiento de las moléculas de una sustancia de una zona de mayor concentración a otra. Por ejemplo, al arrojar una gota de tinta a un vaso de agua, la tinta se irá diluyendo en el agua. Los modelos de difusión toman como entrada imágenes en

este estado, habiendo difuminado la imagen con ruido, e intentan revertir el proceso para llegar a una imagen sin ruido [14]. Este tipo de modelos se utilizan mucho para la generación de imágenes. Algunos ejemplos son Stable Diffusion [3], Dall-E [15], Midjourney [16]… En nuestro caso hemos elegido Stable Diffusion después de un estudio del estado del arte, ya que existen herramientas basadas en Stable Diffusion en las que el proyecto se apoya.

2.2 Comunicando Unity con el modelo de Stable Diffusion

Esta primera parte del desarrollo implica, básicamente, la instalación y configuración de los dos proyectos mencionados anteriormente:

2.2.1 Stable Diffusion WebUI

Este es el primer paso en el desarrollo del proyecto. Siguiendo las instrucciones del repositorio de GitHub, se realiza una primera instalación pero los resultados no son muy prometedores. Nos encontramos con muchos errores al ejecutar la aplicación y, tras una investigación entre los *issues* del repositorio, y diferentes discusiones entre usuarios, descubrimos que este proyecto tiene cierta incompatibilidad con equipos que tuvieran tarjetas gráficas AMD. Resulta que el proyecto se apoya mucho en la tecnología CUDA [17]. Esta tecnología, propietaria de Nvidia, sólo está presente en tarjetas gráficas de su marca.

Encontramos una solución [18], que consiste en redirigir la mayor carga de trabajo al procesador en vez de a la tarjeta gráfica. Esto, si bien hace que la aplicación funcione como se esperaba, también hace que funcione mucho peor, haciendo que generaciones sencillas llegaran a tomar minutos de ejecución.

Después de este contratiempo, decidimos cambiar el entorno de trabajo y continuar el desarrollo en un ordenador equipado con una tarjeta gráfica de Nvidia, para evitar mayores complicaciones y favorecer la compatibilidad con la aplicación.

Teniendo esto en cuenta, solo queda hacer la instalación como se nos indica en el propio repositorio:

1. Instalar Python 3.10.6
2. Instalar git
3. Clonar el repositorio utilizando git
4. Descargar un modelo de Stable Diffusion: hay dos formas en las que podemos encontrar esto, un archivo .ckpt o un archivo .safetensors. Estos dos son bastante parecidos, con la única diferencia de que .ckpt, en algunos casos muy concretos, puede contener código malicioso si se obtiene de fuentes no fiables. Una página muy buena para descargar estos *checkpoints* es CivitAI [19]. Aquí se encuentran tanto modelos como LoRA (Low-Rank Adaptation), que son básicamente extensiones que se pueden utilizar para influenciar las generaciones de otros modelos. Aquí una pequeña guía de qué son y cómo utilizarlos en *Automatic1111* [20]. Descargaremos nuestro archivo .ckpt o .safetensors y lo colocaremos dentro del directorio *models/Stable-diffusion*.

Después solo quedaría ejecutar el archivo webui-user.bat y empezar a utilizar la aplicación en la web. Pero en nuestro caso hace falta hacer un cambio más. Deberemos editar este mismo archivo y añadir la opción --api a la lista de argumentos. Ahora, al ejecutar la aplicación, a parte de montarse la interfaz web también se hará disponible la API para hacer las peticiones.

Ahora, para poder empezar a aprovecharnos de esta API dentro de Unity, pasamos a ver:

2.2.2 AIConnectors

Este proyecto proporciona segmentos de código que facilitan la conexión con la API de *Automatic1111* desde Unity. En concreto hay dos directorios. Uno (*GlobalUse*) contiene código común y utilidades en las que se apoya el resto del código. En el directorio *ImageAI* es donde encontramos las funcionalidades principales.

A continuación [Figura 2.2] vemos el código principal que vamos a utilizar nosotros.

```
public IEnumerator GetImage(string prompt, System.Action<Texture2D> callback,
bool useCache = false, int width = 512, int height = 512, int steps = 50,
int promptStrength = 7, int seed = -1, byte[] image = null, byte[] mask = null,
string cacheKey = null, bool tiling = false, float denoisingStrength = 0.75f,
string negativePrompt = null)
{
    ImageToImageAIParams aiParams = new ImageToImageAIParams()
    {
        prompt = prompt,
        width = width,
        height = height,
        seed = seed,
        promptStrength = promptStrength,
        steps = steps,
        tiling = tiling,
        negativePrompt = negativePrompt,

        initImages = image != null ? new string[] { ImageAIHelper.ImageBytesToDataString(image) } : null,
        mask = ImageAIHelper.ImageBytesToDataString(mask),
        denoisingStrength = image != null ? denoisingStrength : 0f
    };
    return GetImage(callback, aiParams, useCache, cacheKey);
}
```

Figura 2.2: Función *GetImage()* de AIConnectors

Esta es una función que construye todos los parámetros utilizando los que le enviemos como entrada (si no los recibe, los pone a un valor por defecto establecido). Después, hace una llamada extra a otra función, que es la que hace la petición web como tal y el decodificado de la imagen resultante. Si nos fijamos, podemos ver que uno de los parámetros que recibe es una función *callback*. En esta función es en la que haremos uso de la textura resultante.

A continuación [Figura 2.3] vemos cómo estamos utilizando en el proyecto esta función. Vemos que en nuestra callback hay varios pasos a seguir para utilizar la textura. Hacemos la llamada a GetImage pasándole un prompt (esto es, la frase que describe la imagen que queremos generar), un tamaño, un negative prompt (una frase describiendo lo que no queremos ver en la generación) y unos steps: el número de iteraciones que queremos que se utilicen para la generación (a más pasos, más calidad).

```
StartCoroutine(
    imageAI.GetImage(prompt, (Texture2D texture) =>
    {
      try
      {
        Debug.Log("Done.");

        if (SpriteOrTexture(targetObject))
        {
          if (removeBg)
          {
            Color fillColor = new Color(0f, 0f, 0.2f, 0f);
            ImageFloodFill.FillFromSides(texture, fillColor,
                threshold: 0.075f, contour: 5f, bottomAlignImage: true);
          }
          targetObject.GetComponent<SpriteRenderer>().sprite =
          Sprite.Create(texture, new Rect(0, 0, texture.width, texture.height), Vector2.one * 0.5f);
        }
        else
        {
          Material tempMaterial = new Material(targetObject.GetComponent<Renderer>().sharedMaterial);
          tempMaterial.mainTexture = texture;
          targetObject.GetComponent<Renderer>().sharedMaterial = tempMaterial;

        }

        StoreNewTexture(texture);
      }
      catch (System.Exception e)
      {
        Debug.LogException(e);
      }
    },
    useCache: false,
    width: sizeX, height: sizeY,
    steps: steps,
    negativePrompt: negativePrompt
));
```

Figura 2.3: Función GetImage() de AIConnectors

Nuestra función de callback recibe la nueva textura y la aplica al objeto que hayamos elegido como objetivo, ya sea 2D (es decir, tiene agregado una componente SpriteRenderer de Unity) o 3D (con lo que tendrá un componente Renderer de Unity).

Además, vemos que en el caso de que sea 2D, damos la opción a hacer un pequeño retirado del fondo (se escanea la imagen y se elimina el color negro). Para utilizar esto correctamente, deberemos haber indicado a Stable Diffusion que genere una imagen con fondo negro.

Esta llamada la hacemos dentro de uno de nuestros scripts principales, llamado TextureManager.cs. Este script será el encargado de gestionar la generación de

texturas para nuestros objetos y, más adelante, del versionado de las texturas.

2.3 Unity UI Builder, construyendo interfaces customizadas

Para convertir este proyecto en una herramienta como tal, se plantean varias opciones. El objetivo es que el resultado sea “portable”. Es decir, que se pueda llevar a diferentes proyectos con facilidad para utilizarse.

Para esto se hacen varios planteamientos, cada uno con sus pros y contras. Al final, se ha tomado la decisión de dejarlo todo en forma de *scripts* de código que puedan ser importados fácilmente. Ahora bien, queda resolver la problemática de la interfaz: cómo un programador Unity puede hacer uso de la herramienta de forma cómoda.

Para esto, decidimos crear componentes de Interfaz de Usuario customizadas para los diferentes scripts.

Para esto, se realiza una investigación sobre cómo crear Interfaces Customizadas, y se llega a la conclusión de que la mejor alternativa es utilizar el *Unity UI Builder Framework* [21]. Un set de herramientas incluido en las últimas versiones de Unity. Esto proporciona una interfaz donde podemos diseñar las Interfaces, cogiendo elementos y arrastrándolos para colocarlos. También podemos darle estilo a los componentes individuales, o incluso de forma general a “clases” que podemos aplicar luego a los componentes, muy al estilo del lenguaje *css* [22].

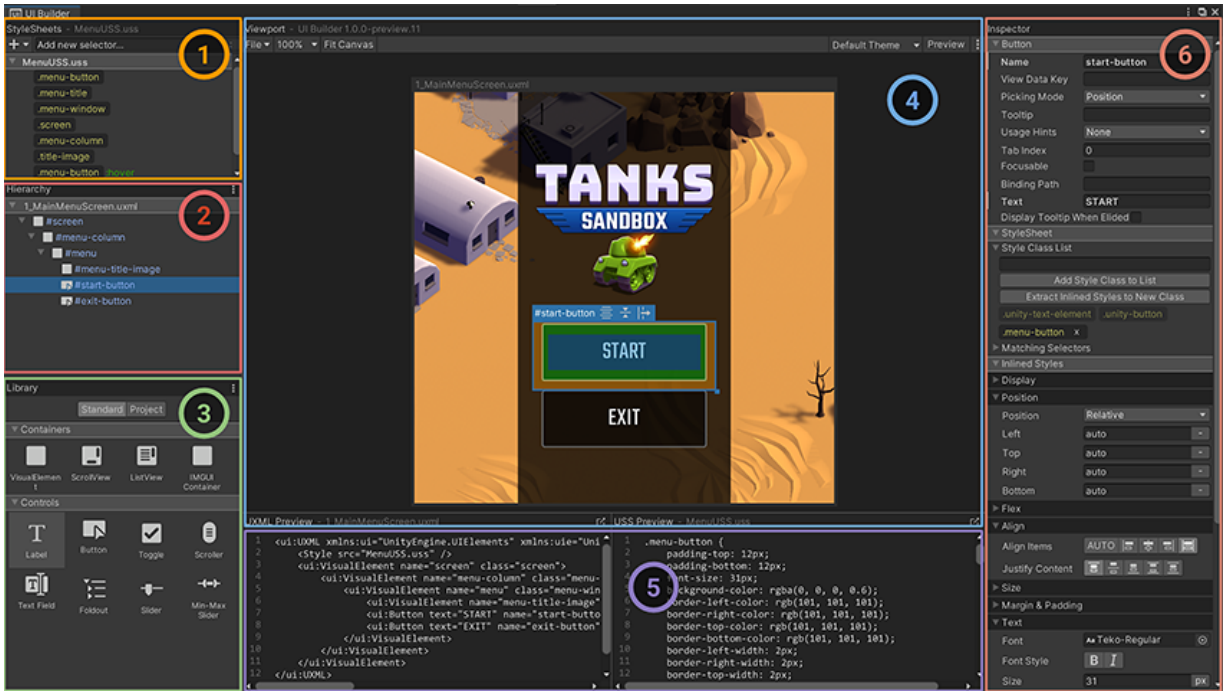


Figura 2.4: Interfaz de Construcción de *UIs* de Unity

Esta [Figura 2.4] es la ventana que contiene toda la interfaz al respecto. En este caso, vemos como la interfaz que se está desarrollando es para utilizar en el juego, pero esta misma funcionalidad también se puede utilizar para customizar las interfaces propias del editor.

- El apartado 1 es donde se pueden añadir y reordenar las hojas de estilo uss que comentábamos anteriormente.
- El panel 2 es la representación jerárquica de los elementos que conforman la interfaz que estamos diseñando.
- El panel 3 contiene todos los elementos prefabricados que podemos utilizar y añadir a la interfaz, arrastrando directamente desde ese panel.
- El panel 4 contiene una previsualización de la interfaz que estamos diseñando.
- El panel 5 contiene previsualizaciones del código tanto UXML (que representa la estructura de la interfaz) como USS (que representa el estilo) de la interfaz que estamos desarrollando.
- El panel 6 es el inspector. Para cada componente de la interfaz hay una serie de propiedades y estilos que podemos modificar en este panel.

Al terminar de diseñar una interfaz aquí, el resultado está recogido en un archivo UXML y, opcionalmente (si hemos creado una hoja de estilos propia para la interfaz), un archivo USS, que Unity interpreta para construir las interfaces [23].

El resultado final de la interfaz desarrollada para el apartado de las generaciones se ve así: [Figura 2.5]

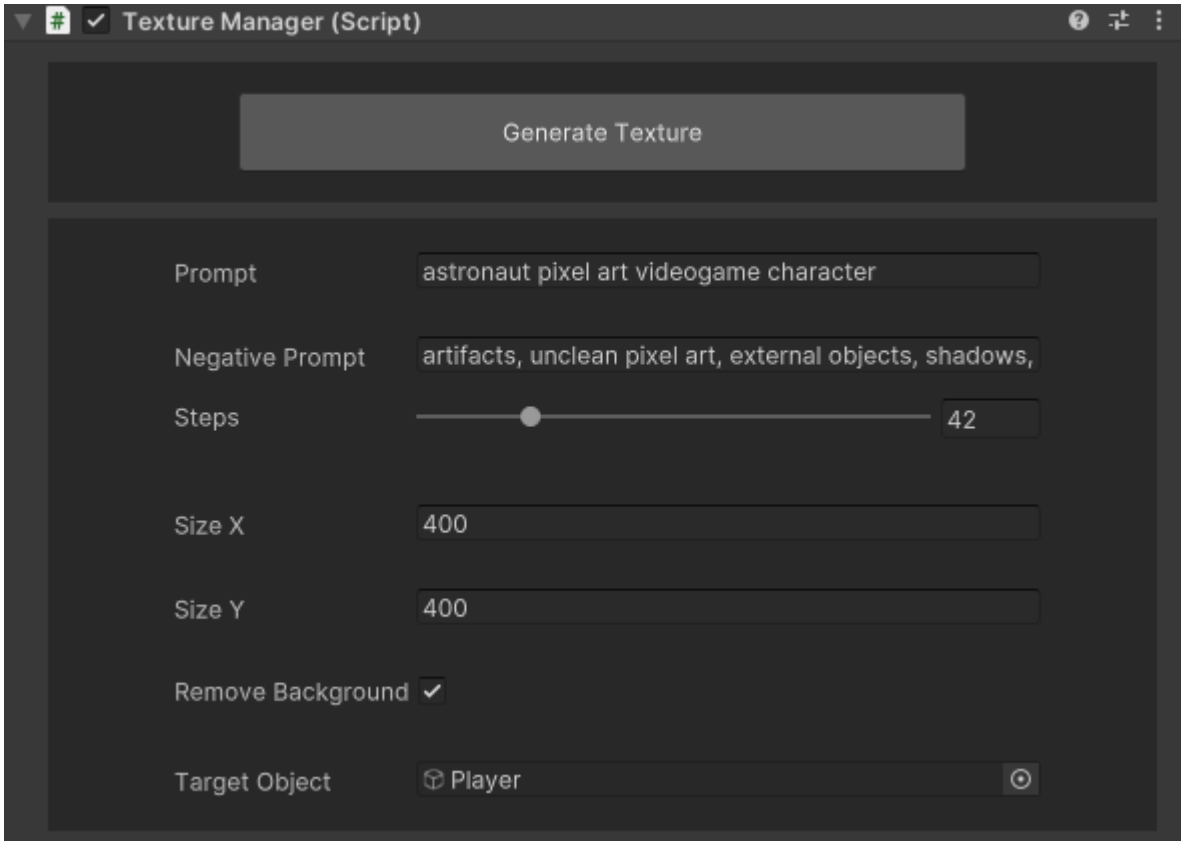


Figura 2.5: Interfaz desarrollada para el script de Texture Manager

Además de diseñar la interfaz como tal, tenemos que hacer un emparejamiento entre los campos y los valores presentes en el script. Para esto, se utilizan los *binding path*. Para cada componente, esto se encuentra en el panel 6. Ahí, asignamos al *binding path* el mismo nombre que tenga nuestra variable pública en el script y se hará el emparejamiento. Por ejemplo, para la variable *targetObject* que se utiliza en el script de *TextureManager.cs*, tendremos que asignarle el *binding path* correspondiente a la componente de selección de objeto de nuestra interfaz.

Ahora bien, queda asignar este código *UXML* al script para el cuál queremos que se sobreescriba la interfaz por defecto. Para esto, bajo el directorio *Editor*, declaramos el script *TextureManagerEditor.cs*. Es importante localizarlo en este directorio porque, por su nombre, es omitido al construir versiones de videojuegos. Por tanto, todo este contenido que solamente atañe al editor de Unity no se comparte.

El script [Figura 2.6] define una función *CreateInspectorGUI*, que se llama al dibujar la interfaz en el editor.

```
#if UNITY_EDITOR
[CustomEditor(typeof(TextureManager))]
public class TextureManagerEditor : Editor
{

  public VisualTreeAsset m_UXML;

  public override VisualElement CreateInspectorGUI() {

    var root = new VisualElement();
    m_UXML.CloneTree(root);

    TextureManager m_TextureManager = (TextureManager)target;
    root.Q<Button>("GenerateButton").clicked += () => m_TextureManager.Generate();
    root.Q<Button>("ResetButton").clicked += () => m_TextureManager.ResetTexture();
    root.Q<Button>("SaveSceneButton").clicked += () => m_TextureManager.AddSceneVersion();
    root.Q<Button>("ResetSceneButton").clicked += () => m_TextureManager.RestoreSceneVersion();
    root.Q<Button>("FlushObjectButton").clicked += () => m_TextureManager.FlushObjectTextures();
    root.Q<Button>("FlushSceneButton").clicked += () => m_TextureManager.FlushSceneTextures();

    return root;
  }
}
#endif
```

Figura 2.6: Código de creación de la interfaz

Podemos observar la directiva de tipo #if, que hace que este script solo se ejecute si estamos en el editor de Unity. También vemos como se define que esta clase representa un editor customizado para nuestros script de tipo TextureManager.

La única variable pública que tiene, m_UXML, es el archivo de tipo UXML que generamos anteriormente al diseñar la interfaz. Luego, la función principal genera un objeto de tipo *VisualElement*, llamado *root*, que representa de forma última la interfaz a

aplicar en el editor para este tipo de scripts.

Vemos cómo se obtiene el script *TextureManager* del mismo objeto, y se asignan a los botones las funciones correspondientes de este script. Aquí [Figura 2.7] vemos un resultado de esta interfaz en Unity, siendo utilizada para texturizar un objeto.

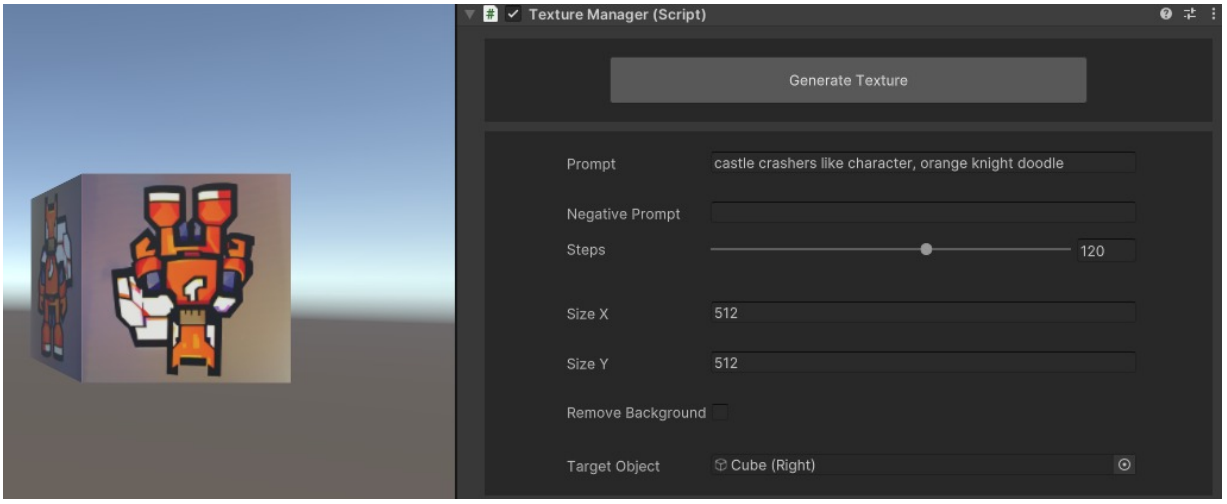


Figura 2.7: Interfaz de texturizado y objeto recién texturizado con parámetros de entrada

Además de esta interfaz, que ya está montada y funcional, se plantea generar una ventana emergente, o *pop-up*. De esta forma, es más cómodo tener la herramienta siempre a mano mientras se navega por la escena. Para esto, creamos un nuevo *script*, llamado *TextureManagerWindow.cs*, también dentro del directorio *Editor*. Este archivo [Figura 2.8] define una clase que hereda de *EditorWindow*, y define la lógica para generar esta componente de la interfaz.

```
public class TextureManagerWindow : EditorWindow
{
   [SerializeField]
   TextureManager m_TextureManager;

   [MenuItem("Textures/Texture Manager")]
   static void CreateMenu() {
        var window = GetWindow<TextureManagerWindow>();
        window.titleContent = new GUIContent("Texture Manager");
   }

   public void OnEnable() {
    m_TextureManager = GameObject.Find("TextureManager").GetComponent<TextureManager>();
   }

   public void CreateGUI() {
    if(m_TextureManager == null)
      return;

    var scrollView = new ScrollView() { viewDataKey = "WindowScrollView" };
    scrollView.Add(new InspectorElement(m_TextureManager));
    rootVisualElement.Add(scrollView);
   }
}
```

Figura 2.8: Código de creación de la ventana emergente

Vemos como en la primera función, se crea la ventana con título “Texture Manager”. Después, en la función *OnEnable()* (que se llama cada vez que el componente se activa) buscamos el objeto llamado TextureManager y obtenemos, a su vez, el script llamado de igual manera.

Después, en la función *CreateGUI()*, que se llama para generar la interfaz de la ventana, creamos una nueva *scrollView*, que básicamente especifica que su contenido se va a enseñar dentro de una vista donde se puede hacer *scroll*.

Finalmente, esta *scrollView* se añade al elemento visual raíz de la ventana, con todo el contenido de la interfaz propia al script.

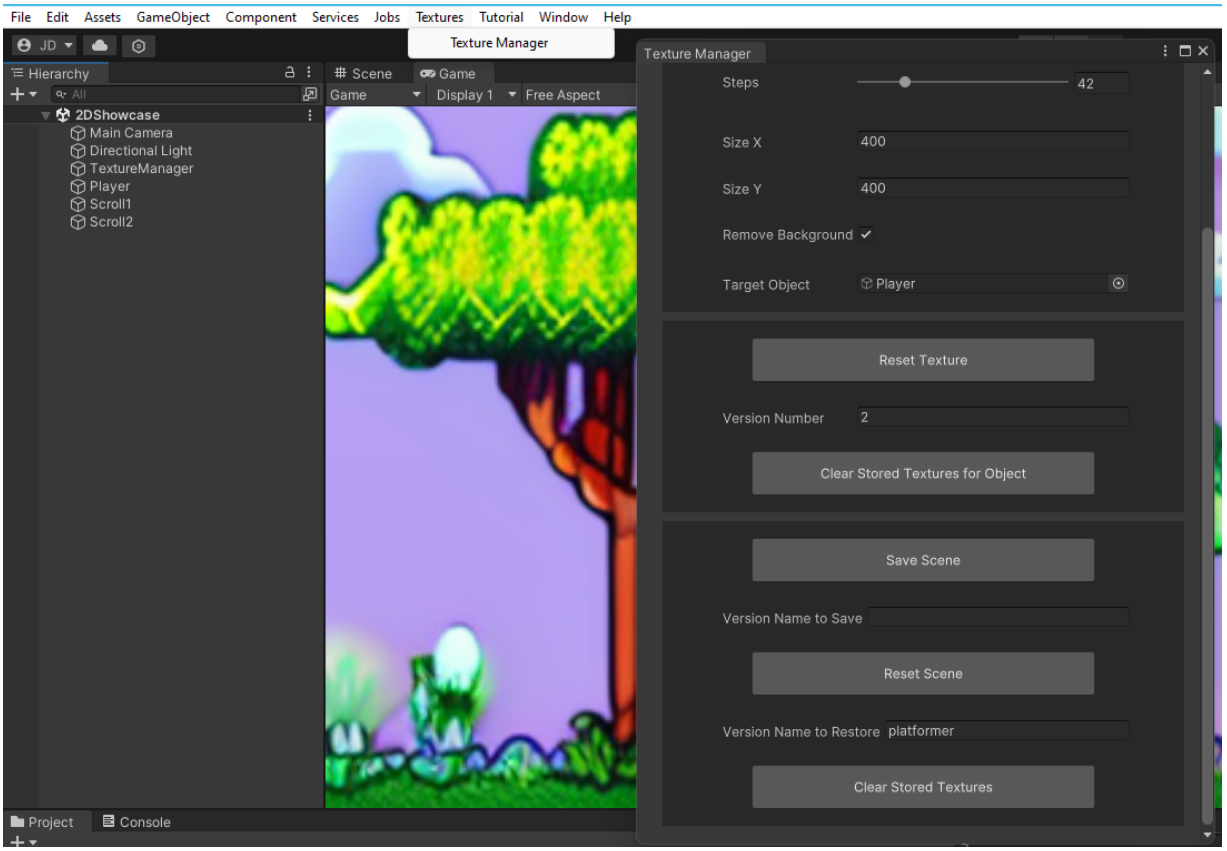


Figura 2.9: Ventana emergente fuera del editor

2.4 Generaciones influenciadas, inpainting y generación de máscaras.

El siguiente desafío que se presenta es intentar aprovechar todas las funcionalidades que proporciona la implementación de *Automatic1111*. Una muy interesante es la función de generar imágenes de forma influenciada. Es decir, además de los parámetros de entrada normales, se proporciona al modelo una imagen de referencia, y un parámetro llamado *denoising strength*. Básicamente, lo mucho (o poco) que queremos que se base en la imagen de origen para generar la imagen nueva.

Esto se puede hacer fácilmente utilizando los conectores que ya usamos. Simplemente, pasamos los dos parámetros extra a la llamada, y texturizamos con la imagen de resultado igual que ya hacíamos antes.

Donde está el verdadero desafío es en el *inpainting*. En Stable Diffusion, el *inpainting*

es un proceso mediante el cual se rellenan las regiones vacías de una imagen con información plausible basada en el contexto visual circundante. En la mayoría de implementaciones, existe una funcionalidad que permite comunicar al modelo ciertas regiones de la imagen original que queremos que sustituya. Esto se suele hacer dibujando encima de la imagen de alguna forma. Aquí [Figura 2.10] un ejemplo de cómo se hace en la interfaz web.

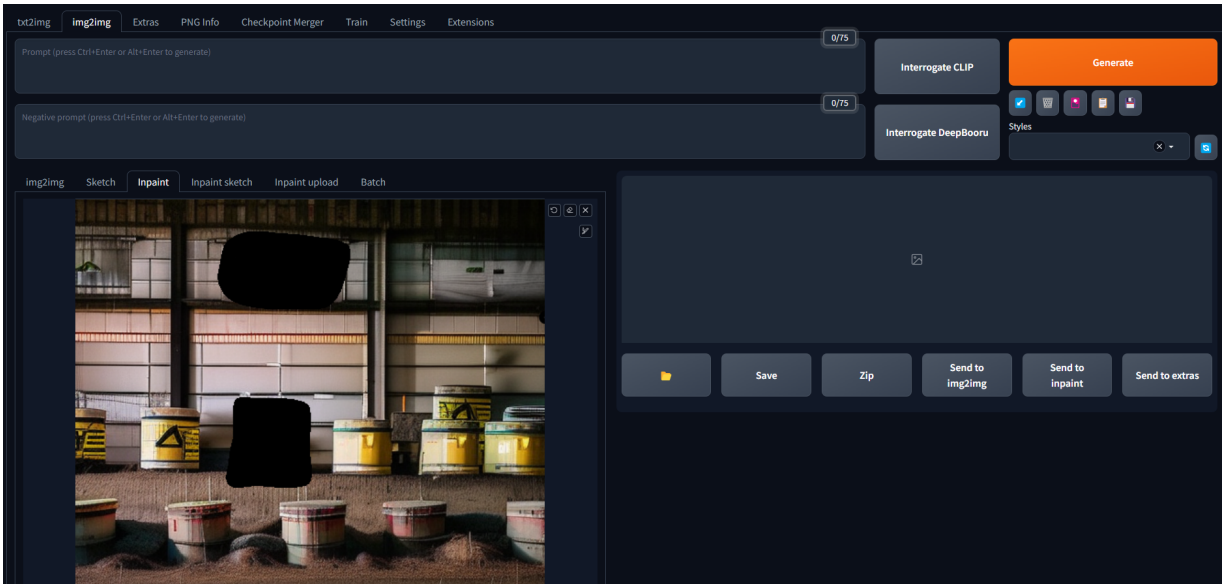


Figura 2.10: Ejemplo de inpainting en la interfaz web

Sin embargo, en Unity, tenemos que ponernos un poco más creativos. Hacer este tipo de dibujado sobre la imagen puede llegar a ser un problema muy complejo y costoso de resolver. Por esto, hemos decidido que vamos a permitir que se dibuje una región rectangular sobre la imagen, dependiendo de cuatro coordenadas. Podremos modificar estas coordenadas y el rectángulo se dibujará sobre la imagen. Cuando estemos contentos con el resultado, se utilizará esta información para hacer la petición a la API.

Básicamente, como veíamos anteriormente, se construye una interfaz personalizada para el script de *InpaintingManager.cs*, que va a contener toda la lógica de las generaciones influenciadas. En esta interfaz, se definen cuatro elementos de tipo *slider*. Es decir, de control deslizante. Estos cuatro representan coordenada de inicio X, coordenada de inicio Y, coordenada de final X y coordenada de final Y. En cuanto a esto,

diseñar la interfaz de forma visual en el *UI Builder* resulta insuficiente para algunas cosas. Por ejemplo, estas coordenadas deben estar recogidas dentro del tamaño de la imagen de origen. En caso contrario, podríamos estar indicando una región externa a la imagen de origen para el *inpainting*. Para especificar esto, lo haremos en *InpaintingManagerEditor.cs*. Como ya vimos en apartados anteriores, aquí podemos hacer cambios más concretos sobre la interfaz diseñada.

Otra funcionalidad que aún no está disponible en la interfaz del *UI Builder* es la de insertar imágenes. Para esta funcionalidad, necesitaremos visualizar la textura del objeto que queremos modificar, para así colocar el rectángulo correctamente.

Esto también lo resolvemos en *InpaintingManagerEditor.cs*. Básicamente, lo que creamos es un “elemento visual” vacío, y cambiamos su imagen de fondo. A esta imagen de fondo le asignamos como valor la textura del objeto habiéndole pintado encima el rectángulo blanco. Esto lo hacemos simplemente recorriendo los píxeles de imagen recogidos por los cuatro puntos del rectángulo y poniéndolos de color blanco.

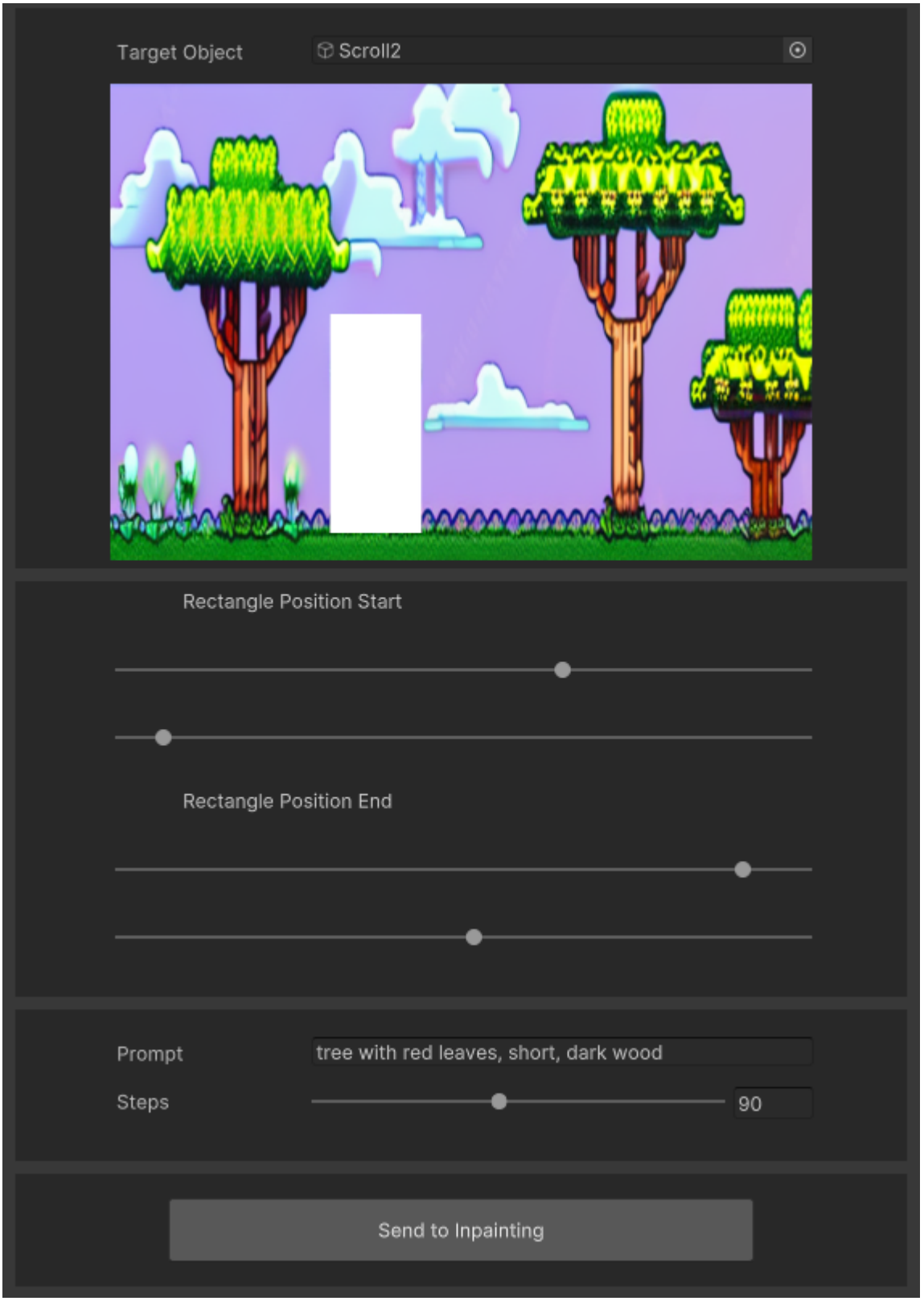


Figura 2.11: Interfaz para el inpainting

Una vez tenemos el rectángulo colocado en la posición deseada, necesitamos encontrar una forma de enviar esa información a la *API*. Esto lo haremos enviando como parámetros dos cosas: en *image* la imagen original, y en *mask* una imagen, del mismo tamaño, que es totalmente negra menos las regiones deseadas que están en blanco.

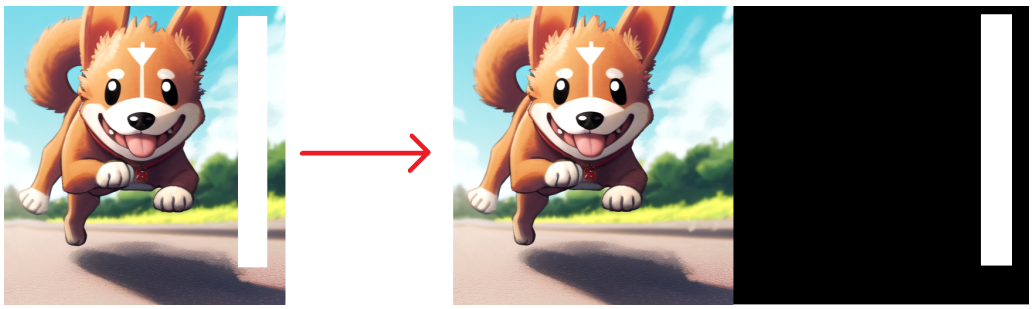


Figura 2.12: Ejemplo de textura y máscara

Aquí [Figura 2.12] vemos qué se envía a la *API*. Obviamente, no se envían las imágenes como tal sino que se envían codificadas.

Al recoger la imagen de resultado, todo lo que vemos en negro en la máscara seguirá intacto, pero lo contenido dentro del rectángulo blanco habrá cambiado según el nuevo *prompt* que hayamos pasado como entrada. Además, todo lo que se haya generado nuevo tendrá más o menos correlación con su alrededor según el parámetro de *denoisingStrenght* (0.0 - 1.0).

2.5 Versionado de texturas y escenas. Persistiendo los resultados.

De los objetivos iniciales, queda por conseguir el tema de gestión de versiones para los objetos. Para resolver esta problemática, vamos a desarrollar un nuevo *script* llamado *TextureVersioningManager.cs*. Aquí, vamos a definir toda la lógica para generar, persistir y restaurar las versiones de los objetos.

Primero, vamos a definir cómo se va a persistir la información. La decisión final es generar archivos en formato *.json* que contengan la información de las texturas. Aquí [Figura 2.13] vemos un ejemplo de archivo que contiene las texturas pasadas de los objetos:

Figura 2.13: Ejemplo de archivo de historial de texturas

Aquí se indican las rutas donde se encuentran, guardados en formato imagen, las texturas que hemos ido aplicando a cada objeto. En nuestro script, definimos una ruta por defecto (MaterialVersions). Bajo esta ruta, se genera un directorio con el nombre de la escena, y otro con el nombre del objeto. Finalmente, en ese último directorio se guardan estas texturas. Esto lo hacemos simplemente cada vez que generamos una textura nueva (es decir, cada vez que llamamos a la *API* para una generación de imagen). Esto se puede observar en la función de *SaveTextureToFile()*.

Toda esta correlación de escenas con objetos, y de objetos con texturas, se guarda en un atributo privado en el script de *versioning manager*. Luego, lo único que queda es monitorizar cambios en ese atributo y escribir esos datos a un fichero cada vez que hay

cambios nuevos.

Cuando iniciemos una escena y el objeto TextureManager se genere por primera vez, este archivo se recuperará del sistema, y se usará para completar este atributo. De esta forma, la información de versiones de texturas persiste entre ejecuciones. Esta lógica se puede observar en las funciones *LoadTextureHistory()* y *SaveTextureHistory()*.

Luego, cuando queremos restaurar la versión de una textura, accederemos a este atributo, obtendremos la ruta donde está guardada la textura que queremos y la restauraremos desde allí. La lógica para esto se encuentra en *ApplyTextureVersion()*.

Además de versionar objetos, queremos ofrecer la funcionalidad de versionar escenas enteras. El caso de uso para esto es, por ejemplo, si queremos generar texturas para un conjunto de objetos en un estilo concreto (por ejemplo: *western*), y luego queremos generar otra vez texturas para todos en otro (por ejemplo: medieval). Luego, gracias a esta funcionalidad, podemos cambiar fácilmente entre las dos versiones y compararlas entre ellas. Esta lógica es bastante similar. Declaramos otro atributo que, cuando decidimos guardar una versión de una escena, registra un nuevo valor que contiene la escena y una lista de objetos con sus versiones. Al restaurar una versión de la escena, se busca y se restauran los objetos uno por uno. Esta lógica se puede observar en las funciones *AddSceneVersion()* y *RestoreSceneVersion()*.

Para no separar mucho la lógica completa del proyecto, se ha tomado la decisión de añadir toda esta funcionalidad de versionado a la misma interfaz que utilizábamos para las generaciones de texturas. Para esto, debemos referenciar las funciones de *TextureVersioningManager.cs* desde *TextureManager.cs*. Así, en la interfaz del segundo podemos hacer llamadas a funciones del primero. Aquí [Figura 2.14] tenemos la interfaz resultante, mezclando las funciones de generación y versionado.

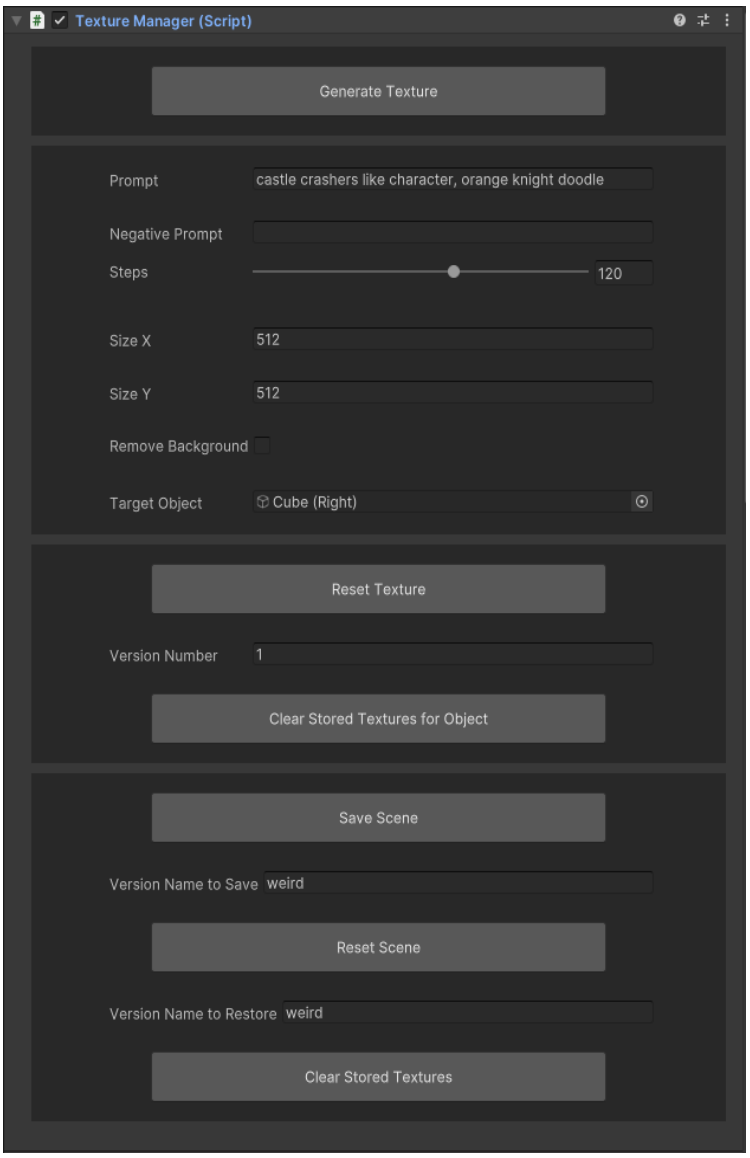


Figura 2.14: Interfaz completa con versionado de texturas y escenas

Capítulo 3 Preparación de la demo / caso de uso

Por último, vamos a preparar unos pequeños ejemplos de casos de uso para nuestra herramienta. Hemos preparado dos escenarios, uno en dos dimensiones y otro en tres dimensiones.

3.1 Escenario en 3D

Para el escenario en tres dimensiones hemos implementado un escenario muy sencillo, con un suelo y paredes [Figura 3.1]. A cada una de estas partes se les proporciona una textura utilizando nuestra herramienta. Además, se posicionan unos objetos que representan a personajes enemigos. A estos también se les da una textura de tipo *sprite*. Es decir, bidimensional. Para que esto se desenvuelva bien en un entorno 3D, hemos hecho que los enemigos se giren siempre de cara al jugador. Este comportamiento recuerda a algunos de los primeros videojuegos en 3D, como el *Wolfenstein 3D*, donde los *sprites* también giraban para estar de cara al jugador.

Figura 3.1: Escena tridimensional con paredes, suelo y enemigos texturizados

Para los controles de primera persona, hemos utilizado unos *asset* públicos muy básicos. Simplemente utilizan una componente de *CharacterController* y una cámara que sigue al jugador para implementar unos controles sencillos, pero efectivos para lo que queremos, que es simplemente navegar la escena.

A continuación, [Figura 3.2], vemos otra escena en espacio tridimensional, también con algunos enemigos, fondo y suelo, y algunas variaciones de la misma [Figura 3.3, Figura 3.4] que hemos generado utilizando la funcionalidad del versionado de escenas.

Figura 3.2: Segunda escena en 3D, texturizada con temática *western*

Figura 3.3: Segunda escena en 3D, texturizada con temática pirata

Figura 3.4: Segunda escena en 3D, texturizada con temática *zombie* / postapocalíptica

Figura 3.5: Escena de la figura 3.4 tras generar una casa de fondo con el *inpainting*

3.2 Escenario en 2D

Para la escena en dos dimensiones, se ha decidido mostrar un poco cómo se podría integrar nuestra herramienta en un videojuego sencillo atendiendo al tema de sobreposición de scrolls. Esto se ve mucho en videojuegos de plataformeo en dos dimensiones, como por ejemplo *Hollow Knight* [24][Figura 3.6].

Figura 3.6: Hollow Knight: escena con sobreposición de planos

Como podemos observar en la imagen, en la escena existen varios planos que se sobreponen, dando un efecto de tridimensionalidad. Algunos planos se presentan como fondo, o *background*, y otros como frente, o *foreground*. Además, en este tipo de aplicaciones, se suele mover los fondos con distintas velocidades, para dar así mayor sensación de profundidad. Esto se conoce como *parallax effect*.

En la siguiente figura [Figura 3.7], vemos la escena preparada con diferentes texturas generadas por nuestra herramienta. Podemos observar todas las componentes distintas, que se mueven a distintas velocidades para emular este efecto *parallax*. Las componentes de fondo (las montañas y el fondo con árboles) se mueven en horizontal utilizando el *offset* o desfase de la textura. Simplemente aplicamos una función que lo va aumentando a medida que pasa el tiempo.

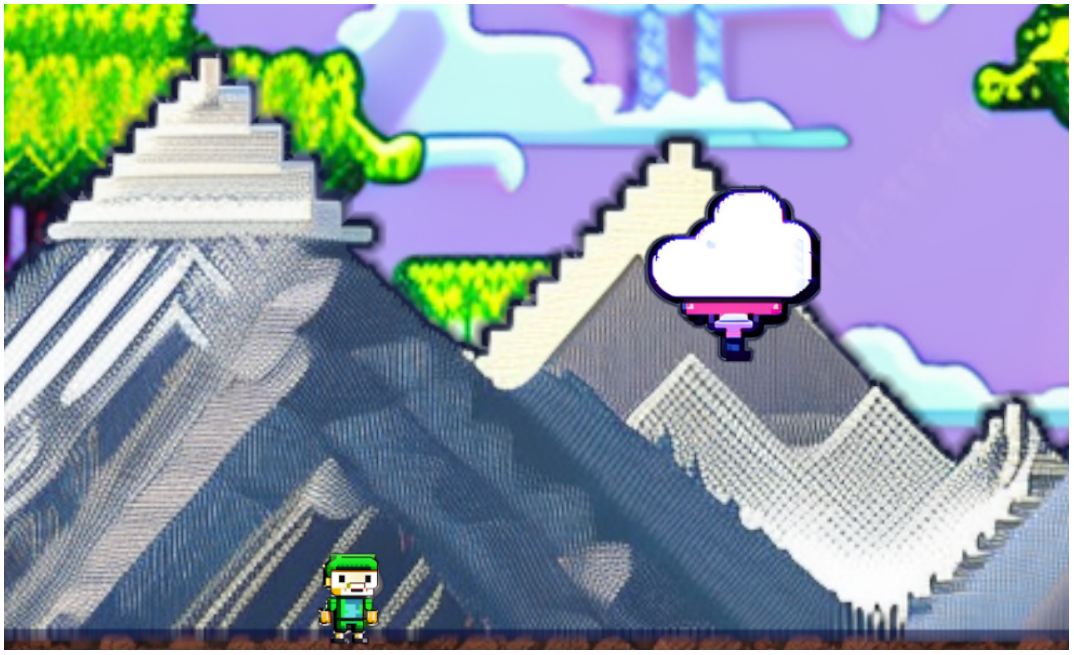


Figura 3.7: Escena en 2D

La nube se mueve gracias a una animación sencilla hecha en el menú de animaciones de Unity (*ctrl + 6* para abrir desde el editor). Además de esto, en esta escena hemos decidido incluir una pequeña muestra para ilustrar el uso del versionado de escenas. Hemos retexturizado el fondo para que sea un planeta alienígena, y la nube para que sea una especie de nave espacial [Figura 3.8]. Así, podemos reutilizar la animación de la nube, que simplemente flota en el aire, para la nave espacial.

Figura 3.8: Misma escena tras restaurar otra versión de las texturas

Capítulo 4 Conclusiones y Líneas Futuras

4.1 Conclusiones

Los modelos de inteligencia artificial han emergido como una fuerza muy influyente en muchas áreas, revolucionando la manera en la que enfrentamos tareas y aumentando nuestra productividad y capacidades. Uno de esos dominios que se han visto afectados por este surgimiento es el del desarrollo de videojuegos. En este trabajo, se ha explorado la integración de los modelos generativos de Inteligencia Artificial con Unity, un motor de desarrollo de videojuegos de los más populares, destacando el inmenso potencial que se puede percibir de dicha combinación.

Sin duda alguna, el arte juega un papel muy importante en el desarrollo de videojuegos. Establece el tono y el ambiente, provee inmersión, y capta a los jugadores. Tradicionalmente, los desarrolladores han tenido que apoyarse en artistas humanos para diseñar cada aspecto de sus mundos. Aunque este acercamiento provee los resultados más prometedores, es un proceso que consume mucho tiempo y recursos. Es aquí donde las Inteligencias Artificiales generativas presentan una alternativa a tener en cuenta.

Al generar resultados de forma tan sencilla, ayudan mucho al prototipado rápido, y se pueden combinar muy bien con el proceso existente para aliviar ciertas cargas, sobre todo de tiempo.

Aún así, no todo es positivo. Hay muchos problemas que se pueden plantear al apoyarnos en este tipo de sistemas, como la fidelidad visual, la coherencia estética, o incluso los asuntos éticos.

En retrospectiva, para más conclusiones podemos hacer una comparativa entre los objetivos planteados al inicio del trabajo y los resultados finales del mismo.

- Se proporciona una herramienta de desarrollo para ayudar al desarrollador Unity a texturizar los objetos de la escena mediante *Stable Diffusion*, mejorando así el proceso de prototipado.
- Además, se ofrece la posibilidad de recuperar, para objetos concretos,

versiones anteriores de las texturas. Este objetivo se amplía para permitir el versionado e iteración para escenas enteras.

4.2 Líneas Futuras

Se plantean diversas formas de mejorar y extender el trabajo realizado. Aquí se incluyen una lista de las más interesantes:

- Mejora de las escenas de caso de uso / inclusión del sistema en un proyecto real: lo primero que se puede ocurrir es el aplicar nuestro sistema de texturizado a un proyecto real, un videojuego entero que se base (bien para el prototipado, o para el producto final) en la herramienta proporcionada.
- Mayor portabilidad: al fin y al cabo, este proyecto, como herramienta de desarrollo, debería ser portable entre proyectos de Unity. Si bien ahora lo es, simplemente teniendo que importar los scripts necesarios, estaría genial mejorar esto para ofrecerlo en la *Asset Store* de Unity, o como módulo de algún tipo, que se instalara solo.
- Mayor integración del modelo: Ahora mismo se proveen ciertas opciones a la hora de hacer las generaciones de imágenes. Si observamos la interfaz web que comentábamos en el capítulo primero, veremos que hay multitud de opciones adicionales que podrían ofrecerse también en nuestra herramienta.
- Integrar otro modelo distinto: para este proyecto, se eligió *Stable Diffusion* porque existía el trabajo anterior, que provee la *API* y todas las funcionalidades. Sería genial poder cambiar el modelo subyacente a todo ese ecosistema para hacer pruebas, ya que se han observado resultados de *Pixel Art* muy curiosos en otros modelos como *Dall-e*.

Capítulo 5 Summary and Conclusions

5.1 Conclusions

Generative Artificial Intelligence models have come up really strong and influential in various areas, revolutionizing the way we tackle tasks and enhancing our productivity and capacities. One of those domains that has been impacted by this, is game development. In this project, we've explored the integration of *AI* generative models and Unity, one of the most popular video game development engines, while highlighting the immense potential one can see in this combination.

With no doubt, we can say that art plays a very important part in game development. It establishes the tone and ambient, provides immersion and captivates players. Traditionally, game developers have had to rely on human artists to design every aspect of the game world. While this approach provides the most promising results, it's a very time consuming and resource intensive process. It's here that generative *AI* models come as a very interesting alternative.

By generating results so simply, they really help when doing fast prototypes, and they can combine really well with the existing process in order to alleviate some workflows.

Nevertheless, everything isn't positive when talking about these. There's a lot of issues still remaining when it comes to using *AI* as part of the design process. For example, there's visual fidelity, aesthetic coherence, or even ethical issues.

In retrospective, in order to express further conclusions, we can compare the initially defined objectives and the project's final state.

- A development tool is provided to help the Unity game developer to texturize objects in the scene using *Stable Diffusion*, enhancing the prototype process by a lot.
- Besides, there's the possibility to get back certain past textures for a given object in the scene. This objective is improved upon by allowing this kind of versioning and iteration on whole scenes.

5.2 Future Improvements

There's a few ways we could improve and extend the current work. Here's a list of some of the most interesting ones:

- Better use-case scenes / inclusion in a real project: the first thing that comes to mind is applying our texturing system to a real project, a full video game that bases its texture generation (for prototyping, or for the final product) on our tool
- Higher portability: this project, as a development tool, should be portable between Unity projects. While it currently is, simply having to import the necessary scripts, it would be great if we could enhance this so that it was downloadable from the Unity *Asset Store*, or as a module of some kind that installed itself.
- Better integration of the model: right now, some options are provided when generating images. If we observe the web interface we saw in chapter one, however, we will see that there's a lot more options we can set there. It'd be great if we could provide those options in our tool.
- Integrating a different model: for this project, we chose *Stable Diffusion* because of the existing project we're based on, that provides the *API* and all other functionalities. It'd be great if we could change around the underlying model of this ecosystem so we could experiment. We've seen better pixel art generation on different models like *Dall-E*.

Capítulo 6 Presupuesto

Se propone el siguiente presupuesto teniendo en cuenta costes tanto de recursos humanos como tecnológicos para el desarrollo del proyecto.

Recursos Humanos:

Tipos	Duración	Coste
Investigación Previa	95h	25 €/hora
Desarrollo del Trabajo	210h	30 €/hora

En cuanto a hardware: dada la necesidad de una tarjeta gráfica NVidia para las generaciones de Stable Diffusion, hemos adquirido un ordenador portátil para el desarrollo de todo el proyecto por un costo de 856.89€.

En un proyecto que utiliza un modelo de inteligencia artificial, es normal ver costos de tiempo de entrenamiento. En nuestro caso, hemos utilizado un modelo pre-entrenado así que no hay gastos por esa parte.

Teniendo esto en cuenta, el coste total del proyecto es de 9,531.89€.

Bibliografía

[1] “Neural Radiance Fields (NeRF) in Unity,” *Unity Forum*. https://forum.unity.com/threads/neural-radiance-fields-nerf-in-unity.1319595/

[2] AUTOMATIC1111, “Stable Diffusion web UI.” Jul. 04, 2023. [Online]. Available: https://github.com/AUTOMATIC1111/stable-diffusion-webui

[3] “Stable Diffusion Version 2.” Stability AI, Jul. 04, 2023. [Online]. Available: https://github.com/Stability-AI/stablediffusion

[4] “What is an API?” https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

[5] P. Lenssen, “AIConnectors.” Jul. 01, 2023. [Online]. Available: https://github.com/JPhilipp/AIConnectors

[6] “Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine,” *Unity*. https://unity.com

[7] “The Wild - VR Collaboration for Architecture & Design Teams.” https://thewild.com

[8] “Unity Powers NASA Virtual Mars Rover Experience.” https://unity.com/our-company/newsroom/unity-powers-nasa-virtual-mars-rover-experience

[9] “3D Animation Software for Film & Television | Unity.” https://unity.com/solutions/film-animation-cinematics

[10] “Visual Studio Code - Code Editing. Redefined.” https://code.visualstudio.com/

[11] “Git.” https://git-scm.com/

[12] “GitHub: Let’s build from here,” *GitHub*. https://github.com/

[13] “What is Trello: Learn Features, Uses & More | Trello.” https://trello.com/tour

[14] J. A. Hernández, “¿Qué son los modelos de difusión en Inteligencia Artificial?,” *Think Big*, Jun. 09, 2023. https://empresas.blogthinkbig.com/que-son-los-modelos-de-difusion-en-inteligencia-artificial/

[15] “DALL·E 2.” https://openai.com/dall-e-2 (accessed Jul. 11, 2023).

[16] “Midjourney,” *Midjourney*. https://www.midjourney.com/home/?callbackUrl=%2Fapp%2F (accessed Jul. 11, 2023).

[17] “CUDA Zone - Library of Resources,” *NVIDIA Developer*, Jul. 18, 2017. https://developer.nvidia.com/cuda-zone

[18] “Install and Run on AMD GPUs · AUTOMATIC1111/stable-diffusion-webui Wiki · GitHub.” https://github.com/AUTOMATIC1111/stable-diffusion-webui/wiki/Install-and-Run-on-AMD-GPUs

[19] “PIXHELL - v 2.0 | Stable Diffusion Checkpoint | Civitai,” Mar. 19, 2023. https://civitai.com/models/21276/pixhell

[20] A. B. Santos, “Stable Diffusion: What Are LoRA Models and How to Use Them?,” *SoftwareKeep*. https://softwarekeep.com/help-center/how-to-use-stable-diffusion-lora-models

[21] U. Technologies, “Unity - Manual: UI Builder interface overview.” https://docs.unity3d.com/Manual/UIB-interface-overview.html

[22] “CSS Introduction.” https://www.w3schools.com/css/css_intro.asp

[23] *Making UI for Games with UI Toolkit ~ Unity 2022 Beginner's Guide to UI*, (2022). [Online Video]. Available: https://www.youtube.com/watch?v=BG6NCgkbbiA
[24] "Hollow Knight," *Hollow Knight*. https://www.hollowknight.com (accessed Jul. 09, 2023).