Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

MÁSTER UNIVERSITARIO EN DESARROLLO DE VIDEOJUEGOS

Trabajo Fin de Máster

Simulador interactivo de principios básicos del sistema inmunitario innato

Interactive simulator of the basic principles of the innate immune system.

Autor Miguel Castro Caraballo

D./Dña. **Rafael Arnay del Arco**, con N.I.F. 78569591G profesor Contratado Doctor adscrito al Departamento de ingeniería informática y de sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

“Simulador interactivo de principios básicos del sistema inmunitario innato.”

ha sido realizada bajo su dirección por D. Miguel Castro Caraballo, con N.I.F. 78537209Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 12/03/2021.

Agradecimientos

A mis padres, por estar pendientes de mi progreso y animarme a continuar.

A mi tutor, Rafael Arnay, por haberme guiado y ayudado tanto en mi TFG como en mi TFM.

Al profesorado del Máster, por haber hecho posible este curso que he podido aprovechar.

A Natalia Arteaga Marrero y Nataliya Lutay, por haberme ayudado de forma desinteresada.

Resumen

Este proyecto tiene como objetivo principal aprovechar la potencia del motor de videojuegos Unity para así ofrecer un simulador o entorno interactivo que fomente y facilite el aprendizaje de aspectos básicos del sistema inmunitario innato del cuerpo humano.

Este simulador ofrece varios modos de ejecución. Uno de exploración, en el que se puede observar la actividad celular en un fragmento de piel (dermis/epidermis), una placa petri, un escenario de ejemplo de virus, y un segundo modo de generación y carga de escenas preparadas cuya utilidad está pensada para docentes que deseen aprovechar la herramienta para mejorar la didáctica de sus clases.

Particularmente, este proyecto permite trabajar con modelos 3D de diferentes células del cuerpo humano, observarlas en distintos entornos, comprobar algunas de sus funciones y conocer, de una forma más dinámica, lo que sucede en el cuerpo humano durante algunos procesos, más allá de los libros de texto o videos renderizados.

El nivel de complejidad o dificultad está pensado para alumnos de enseñanza secundaria y bachillerato. No obstante, dada las características del simulador, cualquier docente puede preparar escenas y preguntas que se adecuen a las necesidades, haciéndolo más fácil o complejo según el público objetivo.

En la medida de lo posible, este simulador se ha basado en información verídica y contrastada que demuestra que lo que se representa es veraz o que lo que se simula es lo suficientemente cercano a la realidad de una forma más abstracta.

Palabras clave: simulador, sistema, inmunitario, innato, unity, videojuegos, docencia

Abstract

The main goal for this project is to leverage the power of Unity's video game engine in order to offer a simulator or interactive environment that enhances and eases the learning process of the basic aspects of the human body innate immune system.

This simulator offers multiple execution modes. An exploration one, where the user may observe cell activity in a skin fragment (dermis/epidermis), a petri dish, or a virus sample scenery, and a second mode of scenery loading and generation, that is thought to be used by teachers who wish to use this tool in order to improve the didactic of their classes.

More over, this project works by using 3D models of different cells of the human body, allowing the user to observe them in different environments, check upon some of their functions and understand, in a more dynamic way, what happens inside the human body as some of those processes take place, further than what's on text books or rendered videos.

The complexity level or difficulty is thought to be meant for high school students. However, given the features of this simulator, any kind of teacher could prepare scenes and questions that match their needs, making it easier or harder depending on the goal audience.

As much as possible, this simulator was made based on true and verified information that demonstrates that what's portrayed here is veracious and close enough to what happens in reality in a more abstract way.

Keywords: simulator, system, immune, innate, unity, video game, teaching.

Índice

Índice de tablas

Índice de figuras

Capítulo 1. Introducción

El sistema inmunitario innato es uno de los complejos mecanismos de defensa del que disponen muchos seres vivos, entre ellos el ser humano, para hacer frente a la invasión de patógenos y, en algunos casos, ciertas situaciones internas que pueden ser perjudiciales para el propio individuo.

Al contrario que el sistema inmunitario adaptativo, no confiere una inmunidad a largo plazo, pero su utilidad radica en ofrecer una temprana y eficaz respuesta ante la presencia de patógenos y todos los procesos que desencadena [1].

El funcionamiento del sistema inmunitario innato es bastante complejo dependiendo de cuánto se desea profundizar en la materia, por lo que dependiendo del público objetivo, los libros de texto y materiales multimedia se suelen adaptar a las necesidades del estudio.

Dentro del sistema educativo de Canarias, encontramos el estudio del sistema inmunitario de forma más generalizada en secundaria [2] y con más profundidad en la etapa de bachillerato [3] donde la materia de estudio se centra más en explicar los mecanismos de la respuesta inmunitaria.

Para diseñar el simulador del sistema inmunitario se usó Unity, un motor de videojuegos multiplataforma gratuito cuya potencia y entorno ayudan a desarrollar de forma más rápida prototipos para videojuegos o, como en este caso, simuladores interactivos donde se usan modelos 3D creados con Blender.

1.1. Antecedentes

Tradicionalmente, el sistema inmunitario se estudia en varias de las etapas educativas obligatorias, pero la forma de estudio suele basarse en la lectura de libros de texto y se hace muy poco uso de materiales interactivos o audiovisuales. El problema que supone usar métodos tan tradicionales de estudio, es que muchos de los procesos solo pueden ser entendidos a través de la comprensión de las palabras y requiere de bastante imaginación para hacerse una idea de cómo el compendio de organismos celulares del cuerpo humano interactúan para hacer frente a diversas tareas. A pesar de que ciertos libros contienen imágenes para representar a las distintas células involucradas, es muy difícil que estas ilustraciones representen gráficamente las acciones que desempeñan las células, la diferencia de escala entre ellas o simplemente su movimiento en el espacio.

1.2. Objetivo General

El objetivo general de este proyecto es desarrollar un simulador interactivo en Unity que permita facilitar un acercamiento al estudio de los principios básicos del sistema inmunitario innato. El principal motivo de atención a esta rama de la inmunología es porque confiere una de las primeras líneas de defensa del cuerpo humano y alberga suficiente complejidad como para necesitar de un simulador que ayude a entender mejor ciertos aspectos de los procesos que comprende.

1.3. Objetivos Específicos

El desarrollo de este simulador tiene en cuenta dos escenarios de uso principales:

1. Escenas de ejecución dinámica donde se observa la actividad celular y cómo responde a algunos eventos. Se da cierto margen para la libre exploración por parte del usuario.

2. Creador de escenas con gestión de preguntas y mensajes para un uso didáctico avanzado.

El propósito del primer apartado es ofrecer al usuario entornos dinámicos donde pueda observar los movimiento de las células, algunos de sus procesos, el efecto que tiene una herida sobre la piel para el sistema inmune y cómo se comportan individualmente las células de forma aislada en una placa petri imaginaria.

En el caso del segundo apartado, el creador de escenas permite que el usuario pueda preparar y diseñar situaciones para así observar el resultado en el transcurso del tiempo. Además, permitirá hacer uso de diálogos interactivos que o bien muestren información al usuario final o solicite alguna respuesta a una pregunta planteada por el profesor.

1.4. Estado actual del tema

La mayor parte de la información acerca del sistema inmunitario está concentrada en libros de texto y no es mucha la cantidad de material didáctico interactivo que se encuentra disponible en la red.

Entre los materiales interactivos del sistema inmunitario que se han encontrado, cabe destacar:

1. *Into the Cell* [4]: extensión de propósito general para ver modelos de diferentes orgánulos celulares e información. No se centra en el sistema inmune, sino en las células animales.
2. *P.tree: Single-Cell Simulator* [5]: simulador de bacterias y virus, pero no del sistema inmune. Está centrado más en la diversión que en el aprendizaje.
3. *Immune Quest* [6]: Immune Quest es un juego dedicado al sistema inmunitario que contiene información muy detallada de distintos procesos. Tiene cuestiones bastante complejas, contiene información muy avanzada y parece ser más indicado para niveles universitarios. Es de pago, pero tiene tres niveles gratuitos. En su contra, los modelos 3D utilizados no representan la apariencia real que tienen los elementos del sistema inmunitario.

1.5. Herramientas utilizadas

Para el desarrollo de este simulador se ha utilizado:

- Unity como motor y entorno de desarrollo para el simulador.
- Git como herramienta de control de versiones.
- Visual Studio C++ 2019 para el desarrollo de código en C#.
- Blender como herramienta de diseño de modelos 3D.

Capítulo 2. Representación del sistema inmunitario en el simulador

Gran parte de la carga de trabajo de este proyecto consiste en estudiar el sistema inmunitario innato y saber cómo trasladar parte de esa información al simulador.

En este capítulo se verá de forma detallada como ha sido el proceso de estudio, la información seleccionada y la adaptación de la información para el simulador.

Tal y como se ha mencionado anteriormente, el sistema inmunitario innato es bastante complejo y comprende una gran cantidad de procesos que, según el nivel de detalle que se quiere alcanzar, puede involucrar cientos de páginas de información. Por ello, dentro de este simulador, la información se mantiene a un nivel de enseñanza de Bachillerato, así que no se entrará en cuestiones complejas como reacciones químicas, proteínas o cualquier información que se centre demasiado en el funcionamiento de un determinado organismo o componente.

2.2. Contenido a representar en el simulador

La información que se espera transmitir con este simulador se puede resumir en:

- Reconocer y observar distintos organismos celulares involucrados en una respuesta inmunitaria innata representativa.
- Reconocer algunos ejemplos de patógenos como las bacterias y los virus.
- Apreciar la piel como mecanismo de defensa primario del cuerpo humano y cómo una herida desencadena algunos procesos dentro de la respuesta inmune.
- Permitir observar algunos comportamientos de los organismos celulares sobre una placa petri, de forma aislada.

2.2.1 Células del sistema inmunitario

La tabla 1 muestra la lista de células del sistema inmunitario incorporadas al simulador con su correspondiente modelo 3D diseñado en Blender.

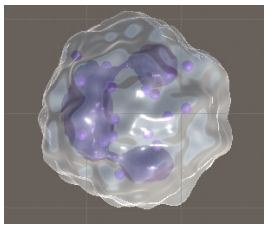
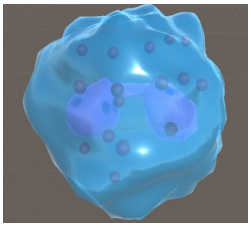
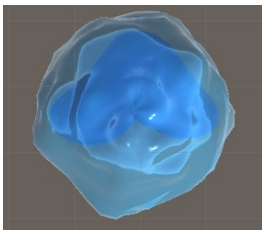
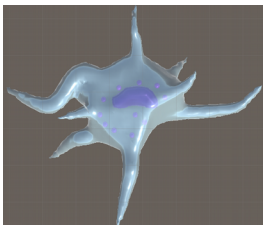
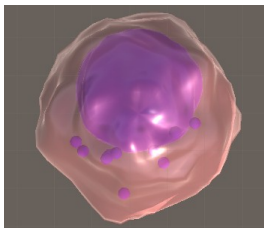
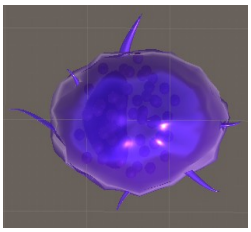
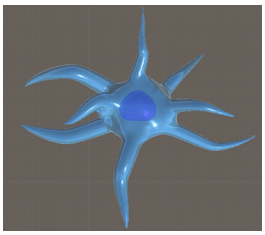
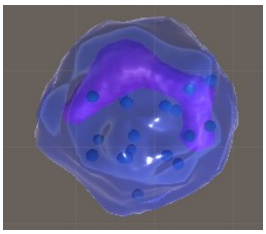
Neutrófilo	Eosinófilo	Monocito	Macrófago
			
Célula NK	**Mastocito**	**Célula dendrítica**	**Basófilo**
			

Tabla 1: Células del sistema inmunitario incorporadas al simulador.

Funcionalidades de células implementadas:

- **Neutrófilo:**
  - Fagocitación de patógenos y células dañadas (la célula es capaz de digerir a diferentes elementos invasores y células dañadas del propio organismo).
  - Degranulación (libera sustancias químicas que son tóxicas para patógenos).
  - Movimiento pasivo y activo (la célula se mueve por atracción química al detectar ciertas sustancias y también tiene movilidad propia).
- **Eosinófilo**:
  - Degranulación (Major basic protein). Esta funcionalidad induce a mastocitos y a basófilos a producir su degranulación.
  - Movimiento pasivo por atracción química al detectar ciertas sustancias.

- **Monocito:**
  - Fagocitación de patógenos y células dañadas.
  - Maduración a Macrófago o Célula Dendrítica.
  - Movimiento pasivo por atracción química al detectar ciertas sustancias.
- **Macrófago:**
  - Fagocitación de patógenos y células dañadas específica a los tejidos.
  - Movimiento pasivo por atracción química al detectar ciertas sustancias.
- **Mastocito:**
  - Mediador de inflamación (libera histamina y citoquinas).

Respecto a las células NK, células dendríticas y basófilos, no se implementan debido a que tienen funcionalidades más singulares o actúan de vínculo con el sistema inmunitario adaptativo.

2.2.2 Células y componentes no pertenecientes al sistema inmunitario

Además de los patógenos, ciertos escenarios del simulador muestran células o componentes que, aunque no tengan mucha influencia en el sistema inmunitario, enriquecen la experiencia del simulador ya sea visualmente o porque forman parte de un proceso mayor en el que también está presente el sistema inmune. La tabla 2 contiene una lista de elementos que se incorporaron al simulador para enriquecer la experiencia de uso.

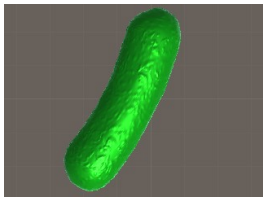
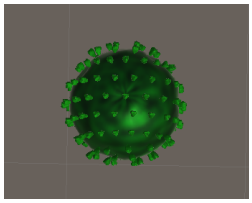
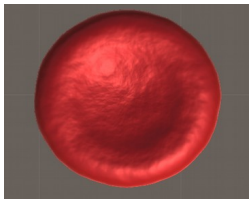
Bacteria	**Virus**	**Glóbulo rojo**	**Plaqueta**
			

Tabla 2: Otros componentes incorporados al simulador.

Capítulo 3. Diseño y desarrollo

En este capítulo se detalla el proceso de diseño y desarrollo del simulador, indagando tanto a nivel de conceptos, funcionalidades y formas de abordar ciertas decisiones de alta importancia como la IA.

3.1. Conceptos globales y funcionalidades

Este simulador ofrece varios modos de ejecución con los que se pretende tener diferentes tipos de acercamientos para el aprendizaje acerca del sistema inmunitario innato.

1. **Modo exploración.**

   Se puede observar la actividad celular en diversas situaciones y en algunos casos interactuar con ciertos elementos. Como su nombre indica, está pensado para una libre exploración y experimentación.

   Se incluye 3 tipos de escenarios:

   Fragmento de piel (dermis/epidermis).

   1. El objetivo principal es mostrar un ejemplo básico de respuesta inmunitaria en la piel mediante la entrada de patógenos a través de una herida.
   2. Se construye un bloque de piel para ver de forma sencilla por qué se considera la piel la primera barrera de defensa.
   3. Se da un primer acercamiento al proceso de heridas en la piel y el correspondiente sangrado.
   4. Se observa insitu la actividad célular y los elementos que circulan por los vasos sanguíneos.

   Placa petri.

   1. Mediante el uso de un panel de instanciación de elementos, el objetivo es que el usuario pueda explorar libremente los elementos que componen el sistema inmunitario. Aunque también se ofrecen algunos otros elementos que interactúan con estos.
   2. Tal y como se indicó en el apartado 2.2.1, algunos de los elementos celulares incluidos en el simulador tienen programadas varias de sus

funciones, por lo que además de observar su representación 3D, también se puede observar algunos de los efectos que puede tener juntar varios de estos elementos en un entorno libre de cuerpos ajenos.

Muestra de virus.

1. El objetivo principal es apreciar a un virus como elemento invasor de tamaño muy reducido y complicado de detectar.

2. En este escenario se ofrece una especie de interacción más o menos guiada para que el usuario se percate por sí mismo del aspecto de los virus en comparación con el resto de elementos.

2. **Modo diseño.**

   Este modo está pensado para los docentes que deseen aprovechar la herramienta para mejorar la didáctica de sus clases. Pueden manipular los elementos que están en los anteriores escenarios y añadir información o preguntas según ocurran ciertos eventos.

Ambas funcionalidades buscan el fin de dar un apoyo en la educación del sistema inmunitario, aunque debido a las características de cada uno, la forma de desarrollarlos varía considerablemente y se verán individualmente en este documento.

3.1.1. Modo de exploración

El modo exploración tiene como objetivo permitir la libre observación de los distintos elementos biológicos que se pueden encontrar en los diferentes escenarios ofrecidos.

A la hora de diseñar este modo, se tienen en cuenta los conceptos que se quiere intentar trasmitir al usuario final de la aplicación a través de un entorno dinámico y no siempre estrictamente guiado, dándole la posibilidad al usuario de observar las situaciones desde diferentes ángulos y en algunos casos con información que complementa la escena que está viendo.

Debido a la naturaleza de las diferentes situaciones propuestas, se pensó que era más conveniente dividir este modo en diferentes escenas y programarlas para que la información no se mezclara con los distintos conceptos que se querían mostrar.

Dentro de estas escenas, gran parte del trabajo se encuentra en la elaboración del propio entorno, programación de elementos específicos e interfaces. La forma de construir estos

elementos se detallará en los respectivos apartados de Modelos, Escenarios e Interfaces gráficas.

3.1.2. Modo diseño

En el modo diseño el usuario puede crear o cargar escenas en las que aparezcan determinados elementos celulares, textos y/o preguntas. Al contrario que el modo exploración, está pensado para un enfoque más académico.

Su funcionamiento es bastante simple y utiliza una interfaz bastante minimalista.

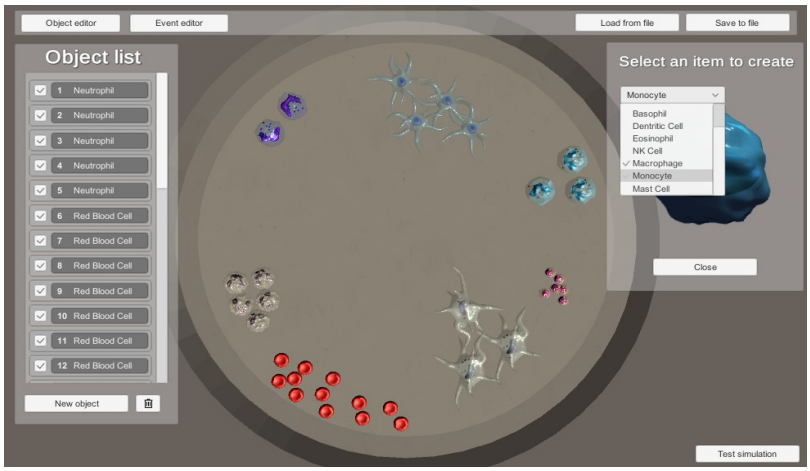


Figura 1: Interfaz del modo diseño

Tal y como se muestra en la figura 1, contamos con un menú superior (principal) en el que tendremos 4 opciones:

1. **Editor de objetos:** Para instanciar los objetos deseados en la escena. Cuenta con un previsualizador del modelo 3D a la derecha y el listado actual de objetos a la izquierda.

2. **Editor de eventos:** Para indicar los eventos que se van a disparar en la escena (texto plano).

3. **Cargar desde fichero:** Carga la información de objetos y eventos desde un fichero.

4. **Guardar en fichero:** Guarda la información de objetos y eventos en un fichero.

Gestión de información y ficheros.

Los objetos y eventos se pueden guardar y cargar a través de un fichero de texto plano que contenga las instrucciones para preparar una escena, de manera que podemos recuperar la configuración de una escena y reproducirla o seguir trabajando en ella.

Los ficheros generalmente se constituyen por dos bloques; “[OBJECTS]” y “[EVENTS]”. El primer bloque representa en cada línea los objetos que apareceran en la escena con sus coordenadas, mientras que el segundo bloque contiene las instrucciones de los eventos, con palabras claves y argumentos para representarlos.

Lista de instrucciones o eventos globales:

- **COUNT** [tipo] [comparación] [operación]

  El evento se activará cuando el conteo de un cierto tipo de elementos coincida con lo indicado en la instrucción.

- **TIME_ELAPSED** [tiempo] [operación]

  El evento se activará cuando haya transcurrido el tiempo indicado en la instrucción desde que se inició la simulación.

- **COLLISION** [tipo1] [tipo2] [operación]

  El evento se activará cuando se haya producido la primera colisión entre dos objetos del tipo indicado en la instrucción.

- **DESTRUCTION** [tipo] [operación]

  El evento se activará cuando se haya producido la destrucción del primer objeto del tipo indicado en la instrucción.

Los argumentos que requieren un “tipo” se refiere al tipo de célula que dispara el evento (neutrófilo, bacteria, etc.).

Las operaciones, si están presentes en las instrucciones, pueden ser las siguientes:

- **INFO** [texto]

  Abre una ventana con información que se muestra al usuario.

- **TOOLTIP** [texto]

Añade un bloque de información en el margen izquierdo de la pantalla que solo muestra el texto cuando el usuario pasa el cursor por encima.

- **QUESTION** [pregunta] [resp1] [resp2] [resp3] [índice respuesta correcta] [explicación]

  Lanza una pregunta al usuario con el texto del primer argumento, seguido por hasta 3 posibles respuestas, continuando por el índice de la respuesta que sería correcta (empezando en 0) y, finalmente, acabando en una explicación opcional sobre la corrección.

En este caso, el mayor coste de desarrollo estuvo en diseñar el sistema de comunicación de mensajes entre los distintos elementos y los eventos.

Se optó por elaborar un sistema de scripts que utilizan delegados y corutinas para comunicar cuando se disparan los eventos y ejecutar las funciones asociadas.

Por ejemplo, el evento *TIME_ELAPSED* es una corutina que mantiene su ejecución en espera hasta que haya trascurrido cierto tiempo, mientras que un evento tipo *COLLISION* es un delegado presente en el script de colisión de un objeto y que ejecutará el código asociado en cuanto se produzca la colisión. Una vez terminado, el delegado elimina la referencia al código a ejecutar.

3.2. Modelado y representación

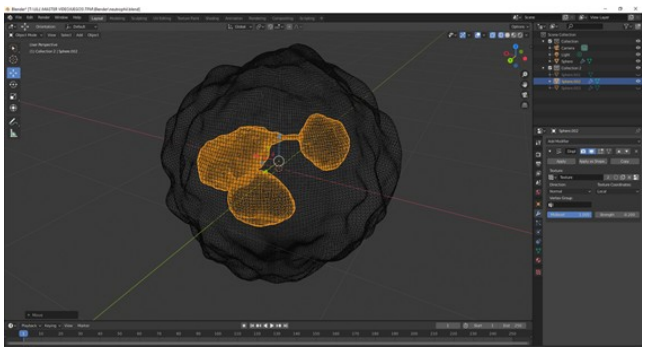


Figura 2: Ejemplo de diseño de modelo en Blender

Una parte significativa de la carga de trabajo para este simulador está localizada en el desarrollo 3D, construcción de escenarios, y técnicas o códigos empleados para poder representar los elementos que permiten ilustrar los elementos del sistema inmunitario.

Naturalmente, la representación de estos elementos requiere de un cierto estudio previo para que lo mostrado tenga una cierta similitud con la realidad, como por ejemplo el diseño del Neutrófilo mostrado en la figura 2, pero en este capítulo solo se detallarán los detalles técnicos y/o gráficos que atañen a este proyecto.

3.2.1. Modelos de células y su adaptación

La gran mayoría del modelado 3D está diseñado en el programa Blender, ya que son modelos muy específicos y su disponibilidad en internet es bastante limitada o requeriría hacer un desembolso económico.

En cuanto a la forma de representar las células y otros componentes en Unity, el proceso requiere una cierta adaptación ya que, en la realidad no tienen el mismo tamaño y es conveniente que, para no confundir al usuario, los modelos se intenten adecuar a un tamaño más o menos realista. Para no complicarlo demasiado, se utilizó una regla (asset) en Unity como guía (ver figura 3) y un script para adaptar los modelos a partir de una referencia.

Tomando como referencia y ejemplo un neutrófilo que mide 12-15 µm en la realidad, se ha hecho que coincida con una medida de 2m sobre la regla de Unity. Es decir, consideramos que 12 µm reales son 2 m (6 µm = 1m) en el espacio de Unity y, al tomarlo como base, tendremos un “1” en los valores de escala del componente Transform.

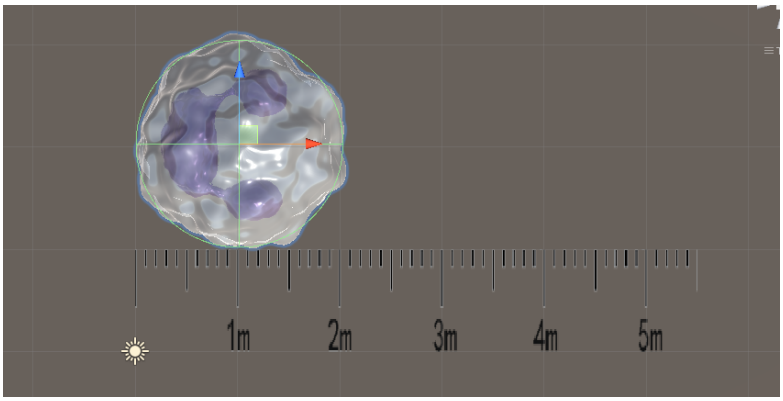


Figura 3: Adaptación y unificación de tamaños de los modelos

Actuando en consecuencia, calculamos el resto de proporciones para otros elementos celulares utilizando la referencia anterior, medimos con la regla y guardamos los valores de escala del componente Transform en un script para poder instanciarlos con una escala adecuada.

Por ejemplo:

Un glóbulo rojo mide aproximadamente 8.2 µm en la realidad, lo cual, comparado con nuestra referencia de 6 µm = 1m , representa un 1.36 en la regla de Unity. Al redimensionar el glóbulo rojo sobre la regla, anotamos los valores de escala (Transform) observados y tendremos la capacidad de instanciarlo siempre correctamente.

La razón de hacer esta adaptación es porque al crear los modelos 3D, algunos terminan midiendo más que otros dentro de la escala propia del programa de diseño y mantener una proporción durante el diseño se hace mucho más costoso. De la manera aquí mostrada, adaptamos los modelos con mayor rapidez en base a la escala de Unity y no a la de un programa de diseño externo.

3.2.2. Optimizaciones básicas

A la hora de modelar varios de estos modelos se tuvo en cuenta detalles técnicos y visuales que pueden afectar al rendimiento del programa, por lo que tanto en el diseño como su adaptación en Unity se procuró seguir una serie de optimizaciones básicas:

- Debido a que la mayoría de modelos 3D son objetos esféricos, podemos aprovechar la situación para usar *colliders* de tipo s*phere* y no tener que recurrir a la propia malla del modelo, esto incurre en un menor cálculo computacional para el motor de Unity al calcular las colisiones.

- Varios modelos contienen muchas esferas pequeñas (gránulos de color púrpura) que pueden incurrir en un aumento considerable del número de vértices por modelo. Considerando que son un detalle muy pequeño, generamos las esferas con un número bajo de vértices para no desperdiciar tiempo de renderizado.

- En el caso de los glóbulos rojos, que es un elemento visualmente muy llamativo (tanto por cantidad de veces que aparece como por forma y color), se recurrió a usar la técnica de *Level of detail* (LOD), con la cual se dispusieron 3 niveles de detalles con diferentes número de vértices (véase la tabla 3). Es decir, cuando la cámara está muy cerca del modelo, se renderiza un modelo visualmente muy atractivo aunque costoso, pero a medida que se aleja, se utiliza un modelo más pobre en vértices para ahorrar tiempo de cómputo de gráficos.

LOD en glóbulo rojo		
LOD 0 = 96420 verts	**LOD 1** = 1566 verts	**LOD 2** = 81 verts

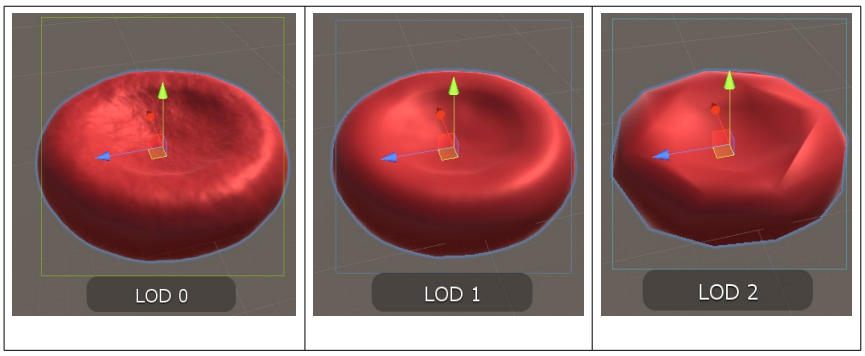


Tabla 3: Diferentes LOD para modelo de glóbulo rojo

3.2.2. Escenarios

En este simulador existen 3 escenarios principales: un fragmento de piel, una placa petri y una sección de vaso sanguíneo aislado para el ejemplo de un virus.

Cada uno de estos escenarios tiene un propósito diferente y se construyen de diferente manera para representar la escena en la que se intentará transmitir algún conocimiento al usuario.

En este capítulo se verá con detenimiento la forma de construir cada uno de estos escenarios, las complejidades de su desarrollo en Unity y formas de abordar sus soluciones.

Sección de piel (dermis/epidermis).

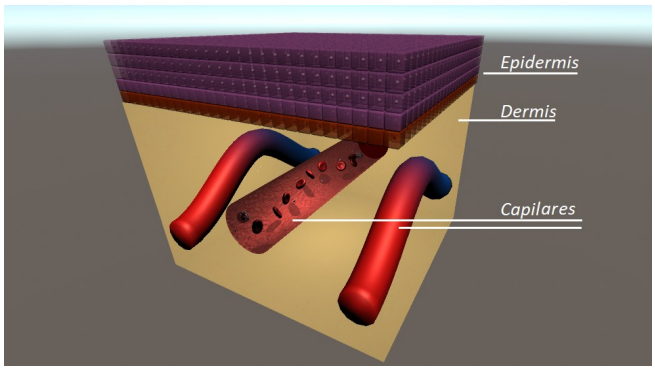


Figura 4: Escenario del bloque de piel en Unity

Sin entrar en detalle en conceptos biológicos, la piel se suele representar como un conjunto de capas que conforman la epidermis (cuya complejidad varía según la enseñanza), seguido de la dermis, donde residen los capilares sanguíneos.

Para intentar representar la epidermis de una forma visual con un cierto parecido a la realidad, se procedió a preparar un script que instanciara una gran matriz de objetos. En la escena final se procede a usar una matriz de 26x25x4 para las capas más superiores más una de 26x25 para la capa inferior. En total son 3250 instancias más sobre el escenario, tal y como se puede ver en la figura 4.

En las primeras pruebas, los FPS del programa caían de 120 a 45, esto es debido a que no se había activado la opción de “GPU instancing” y las llamadas de dibujado no se podían apilar porque los modelos utilizados tenían diferentes materiales (pared y núcleo).

El orden del dibujado de los materiales (render queue) influye en que las llamadas de dibujado se puedan optimizar. Al hacer que tengan un orden específico por cada material, conseguimos que todas las llamadas se puedan apilar, volviendo a conseguir 120 fps.

Con esta fluidez, podemos además aplicar algunos efectos de postprocesado (profundidad de campo, oclusión ambiental, etc.) cuando situamos la cámara completamente fuera del escenario, dentro de la dermis o dentro del propio vaso sanguíneo mediante el uso de volúmenes de postprocesado (ver figura 5).

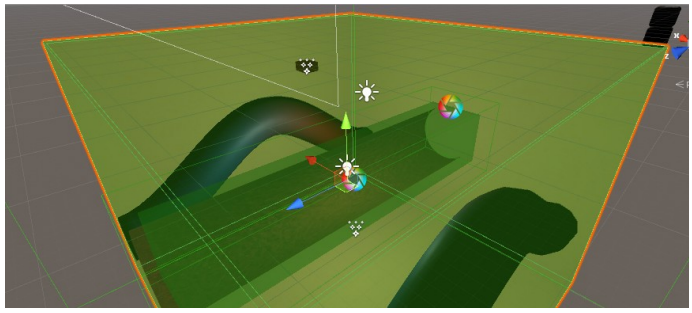


Figura 5: Volúmenes de postprocesado en escenario de piel

En el vaso sanguíneo de referencia (centro de la caja), instanciamos los objetos que simulan el flujo de componentes (células, plaquetas, etc.) mediante un script que gestione los objetos. La importancia de este script consiste en la gestión de recursos. Instanciar y destruir objetos constantemente es mucho más costoso que reutilizar objetos. Así que se procedió a crear un *spawner* que utilizara *pools* para reutilizar los objetos instanciados. De esta manera, cuando

un componente sanguíneo llega al final de la caja, lo devolvemos al pool de objetos y lo marcamos como disponible para su uso.

En este escenario, la cámara está programada para que el usuario la pueda mover libremente por el entorno, dándole la capacidad de exploración que se indicaba en los objetivos del proyecto.

Es interesante remarcar que aquí hay un mecanismo para generar un corte en la piel y observar la cascada de efectos que ello produce a niveles generales (respuesta inmune). El proceso se podría representar con el diagrama de la figura 6:

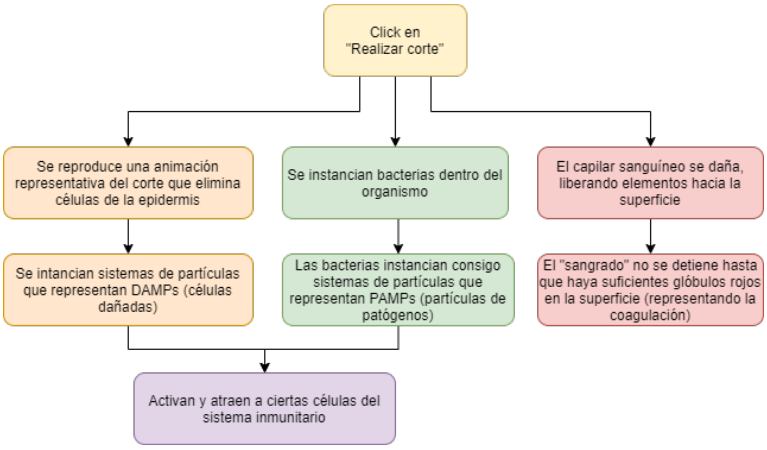


Figura 6: Cadena de procesos al ejecutar corte en la piel

Placa petri

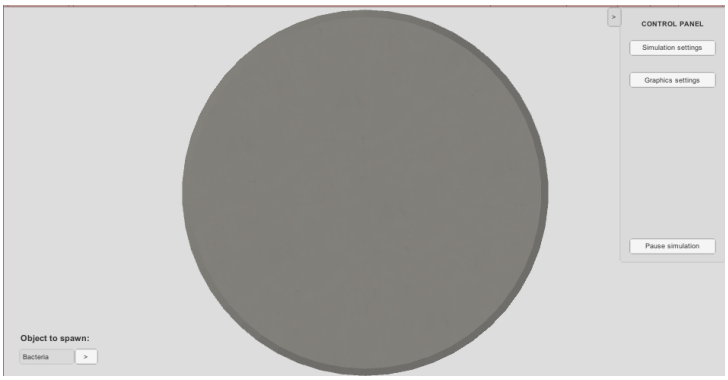


Figura 7: Diseño simple de placa petri

La placa petri está pensada para ser un escenario básico donde el usuario puede instanciar elementos de forma aislada, observarlos, y en algunos casos apreciar comportamientos simulados.

Si nos fijamos en la figura 7, el desplegable inferior izquierda permite seleccionar de la lista el elemento que se quiere colocar. Después, el usuario solo tiene que hacer *click* en el punto donde lo quiere instanciar y el programa lo colocará dentro de la placa petri, previniendo que se instancien elementos fuera de la zona de pruebas.

La gran diferencia de este escenario con los demás es que se utiliza una cámara con vista cenital (simulando un microscopio), permite instanciar elementos libremente y al tratarse de un entorno libre de fuerzas externas (como el flujo de un vaso sanguíneo), se puede observar la actividad con tranquilidad.

Escena de virus

El escenario del virus tiene la peculiaridad de estar construido de una forma interactiva pero más simple y guiada. En concreto, esta vez instanciamos glóbulos rojos con un sistema de partículas pero no el resto de elementos que también viajan por los vasos sanguíneos para no distraer al usuario.

La mayor parte del trabajo de este entorno se encuentra en el uso de la lógica en los scripts y la gestión de información mediante ventanas y textos emergentes.

Manteniendo clara la idea de que el usuario parte de cero, se pretende conseguir hacerle reflexionar y no solo mostrarle una cierta información que pueda descartar fácilmente.

Dado un torrente sanguíneo limpio, se le da la posibilidad al usuario de introducir un virus, tal y como se muestra en la figura 8. Cuando el usuario active el mecanismo de entrada del virus, se le avisará mediante un mensaje de que la operación fue satisfactoria, pero verá que, a pesar de esperar o mover la cámara, no se aprecia ningún cambio a simple vista. Será entonces cuando se revele un *tooltip* que le informe de por qué no ve nada. Esto a su vez, revela un nuevo botón “Find virus” que permite al usuario hacer que la cámara viaje hasta el lugar donde está el virus.

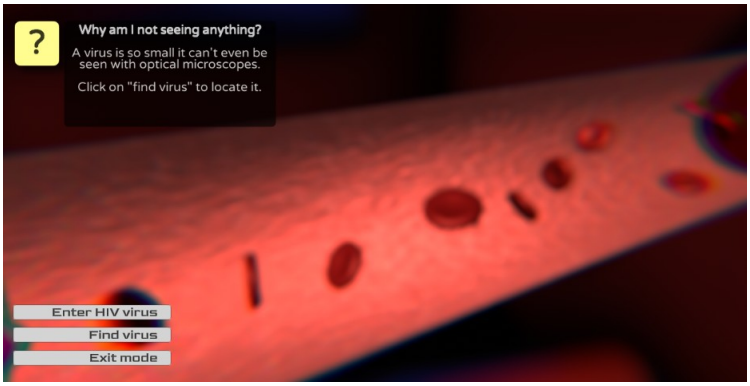


Figura 8: Escenario de virus. Interfaz y ejemplo de tooltip

Con esto, el usuario verá que el virus es tan pequeño que no es tan fácil de encontrar ni siquiera moviendo la cámara con el simulador libremente.

Una vez termina este paso, se le permite comparar el tamaño del virus con otros elementos para que compruebe visualmente lo pequeño que es un virus con respecto a una serie de elementos ofrecidos como referencia (glóbulo rojo, neutrófilo y bacteria).

Por ejemplo, un virus con respecto a una bacteria se mostraría de la siguiente forma:

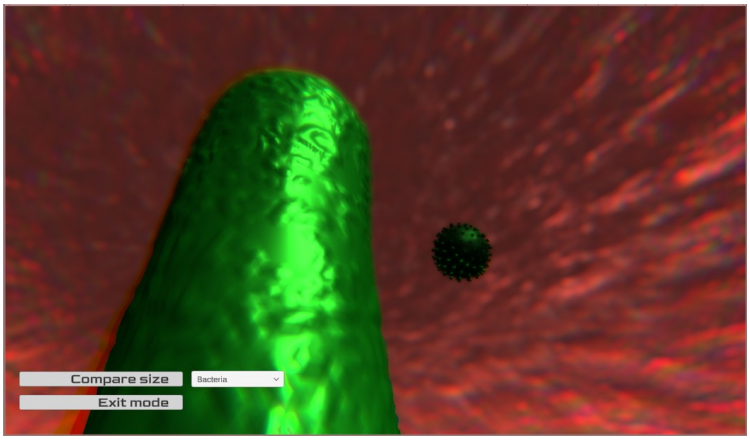


Figura 9: Ejemplo visual de virus vs. bacteria con tamaños algo realistas

3.3. Lógica e Inteligencia Artificial

Cada célula dentro del sistema inmunitario tiene funciones específicas para lidiar con las situaciones adversas que se pueden dar dentro de un organismo. Representarlas todas sería bastante complejo y algunas tienen propósitos que solo se podrían observar mediante su efecto en el entorno, como por ejemplo, el proceso de contracción y dilatación de los vasos sanguíneos ante una herida.

Sin embargo, debido a que muchas células aúnan comportamientos similares, se procedió a generar una lógica base para todas ellas (ver anexo WBC.cs) y luego especificar cualquier posible individualidad en scripts distintos. De esta manera, usando herencia ahorramos bastante código y nos aseguramos de que, si el comportamiento base funciona, lo hará para todas. El diagrama de clases de los mencionados scripts se puede observar en la figura 10.

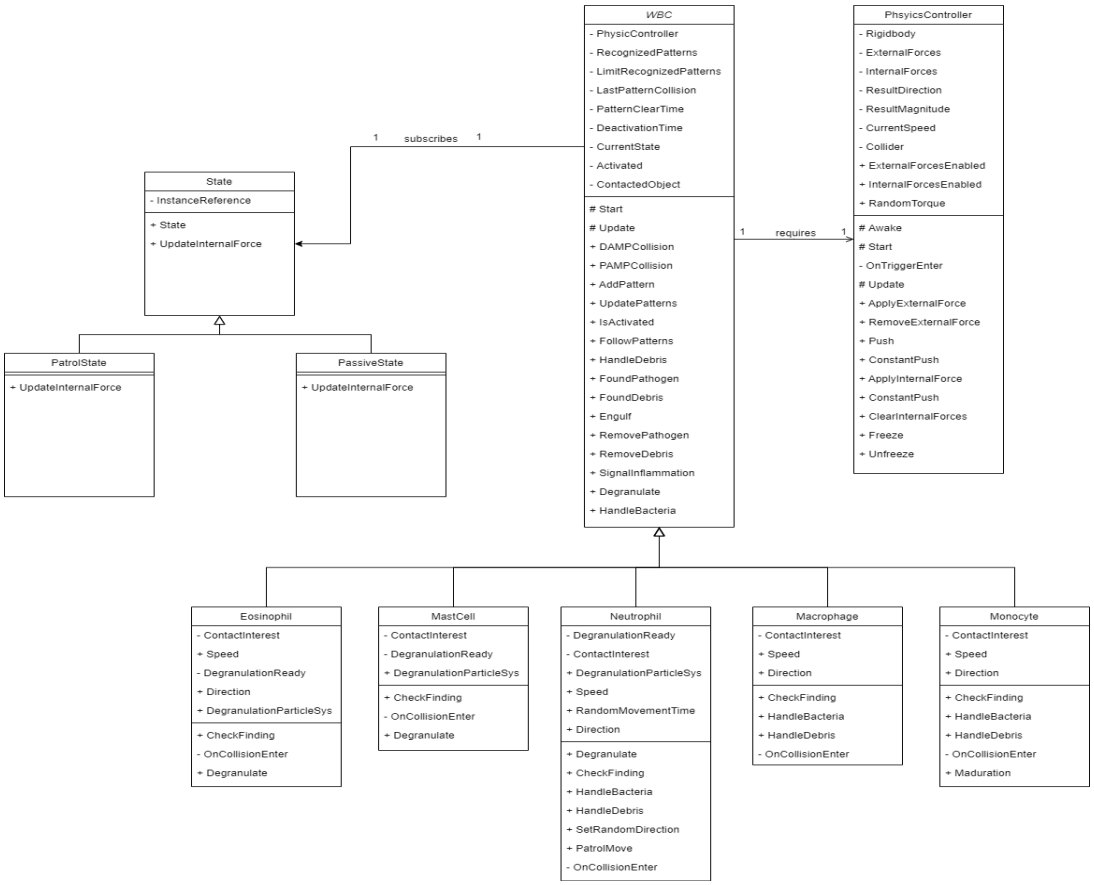


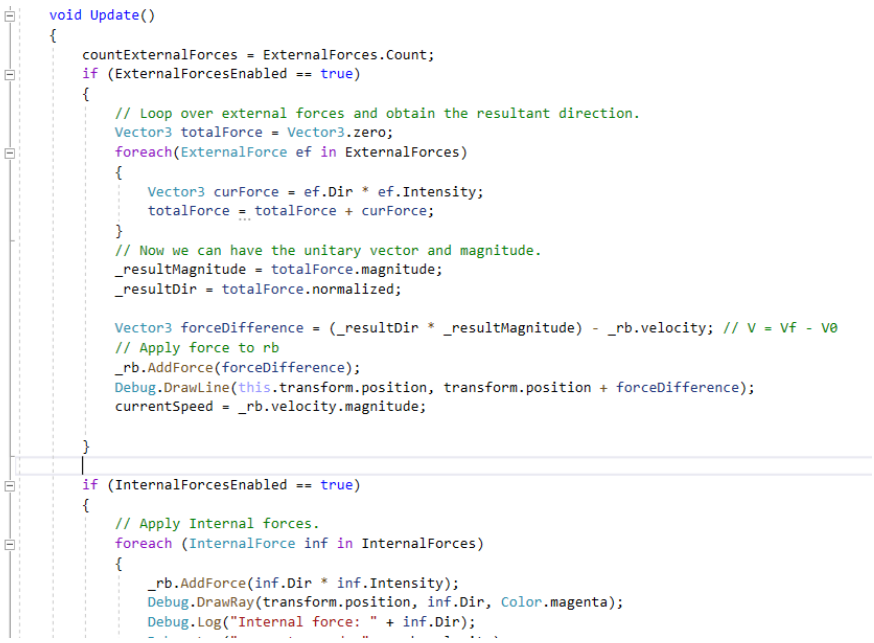
Figura 10: Diagrama de clases

Además de esto, también se utilizan scripts comunes a todos los elementos implementados, donde uno de los más importantes es un controlador de físicas (ver anexo PhysicsController.cs).

Una de las secciones más importantes en el simulador sería la gestión de fuerzas externas e internas.

- Las fuerzas externas son aquellas que modifican la velocidad del rigidbody desde fuera, como por ejemplo, la presión de un vaso sanguíneo.

- Las fuerzas internas, son aquellas que provienen del propio elemento en cuestión, y se utilizan sobretodo cuando existe una atracción por otro elemento, como cuando un glóbulo blanco se siente atraído por una molécula de una célula dañada.



```
void Update()
{
    countExternalForces = ExternalForces.Count;
    if (ExternalForcesEnabled == true)
    {
        // Loop over external forces and obtain the resultant direction.
        Vector3 totalForce = Vector3.zero;
        foreach(ExternalForce ef in ExternalForces)
        {
            Vector3 curForce = ef.Dir * ef.Intensity;
            totalForce = totalForce + curForce;
        }
        // Now we can have the unitary vector and magnitude.
        _resultMagnitude = totalForce.magnitude;
        _resultDir = totalForce.normalized;

        Vector3 forceDifference = (_resultDir * _resultMagnitude) - _rb.velocity; // V = Vf - V0
        // Apply force to rb
        _rb.AddForce(forceDifference);
        Debug.DrawLine(this.transform.position, transform.position + forceDifference);
        currentSpeed = _rb.velocity.magnitude;

    }
    if (InternalForcesEnabled == true)
    {
        // Apply Internal forces.
        foreach (InternalForce inf in InternalForces)
        {
            _rb.AddForce(inf.Dir * inf.Intensity);
            Debug.DrawRay(transform.position, inf.Dir, Color.magenta);
            Debug.Log("Internal force: " + inf.Dir);
```

Figura 11: Sección de script de fuerzas externas e internas

Tal y como se puede apreciar en la figura 11, el script no es complejo de entender pero resulta de evidente utilidad para gestionar cualquier elemento que se pueda incorporar al simulador y requiera de un comportamiento físico similar al de los elementos actuales.

En cuanto a la inteligencia artificial, es conveniente recordar que en este proyecto solo se tratan los principios básicos y no todos y cada uno de los procesos involucrados en la respuesta inmune, así que no todas las células están dotadas de IA.

Para recrear los comportamientos principales y simular la toma de decisión de una célula se decidió utilizar árboles de comportamiento con Panda Behaviour [7], ya que por lo general las funciones se desencadenan en base a condiciones no demasiado complejas.

La forma de abordar la representación del comportamiento de una célula es la siguiente:

1. Entender las funciones de la célula y decidir cuáles se van a representar (más importantes, interesantes, básicas, etc.) en base a un estudio previo.

2. Elaborar un fichero .BT con la lógica que representaría las funciones elegidas.

3. Desarrollar la lógica que permita observar estos comportamientos con cierta facilidad sin tener que esperar demasiado por el simulador.

Para que sea más fácil de ver y entender, pongamos por ejemplo la lógica de un neutrófilo, que es una de las células del sistema inmune más completa:

```
1  // https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4188125/
2
3  tree "Root"
4      repeat
5          mute
6              fallback
7                  tree "Inactive"
8                  tree "Active"
9
```

Figura 12: Panda Behaviour: Estados de un neutrófilo

Los neutrófilos pueden estar activos o inactivos dependiendo de si detectan la presencia de patógenos en el entorno.

```
10  // If no pathogens, handle debris and patrol.
11  tree "Inactive"
12      while not IsActivated()
13          fallback
14              HandleDebris()
15              tree "Patrol"
```

Figura 13: Panda Behaviour: Funciones de neutrófilo al no estar activado

Cuando están inactivos, tomamos como base que los neutrófilos en el organismo simplemente están a la espera de “detectar algo”. Por lo general, si no están atacando patógenos, pueden estar realizando la tarea de limpiar el organismo de restos celulares

(debris). Si no hubiera nada que limpiar, simplemente siguen buscando (movimiento de patrullar).

```
17   tree "Patrol"
18       sequence
19           PatrolMove()
20           Wait 3.0
```

Figura 14: Panda Behaviour: Movimiento exploratorio de neutrófilos

Patrullar, adaptándolo al simulador, se basa en moverse aleatoriamente y esperar 3 segundos antes de realizar alguna otra acción para prevenir que el impulso del *rigidbody* sea constante.

```
25  // While active, try to deal with pathogens first, otherwise just follow traces.
26
27  tree "Active"
28      while IsActivated()
29          fallback
30              tree "DealPathogens"
31              HandleDebris()
32              FollowPatterns()
```

Figura 15: Panda Behaviour: Estado activado de un neutrófilo

En caso de que el Neutrófilo estuviera en estado activo (desencadenado por la detección de un PAMP o DAMP) el comportamiento varía. El movimiento deja de ser aleatorio y se basa en seguir el rastro de las moléculas que lo activaron y tratar a los posibles patógenos encontrados.

Aquí podemos enfatizar lo interesante de adaptar este tipo de situaciones en Unity. Si por ejemplo el estudio de la biología celular nos dice que los neutrófilos se orientan y dirigen hacia la zona donde existe una concentración mayor de moléculas por las que sientan atracción, la forma de adaptarlo puede ser más o menos compleja según la decisión del desarrollador. En este proyecto, la forma de abordarlo con cierta concordancia pero con sencillez, se basa en crear una función que, dependiendo del estado, calcula un promedio de vector de dirección en base a la posición relativa de las partículas que han ido colisionando con el neutrófilo.

En general, como se ha podido comprobar con el ejemplo, el apartado de inteligencia artificial en este proyecto es bastante sencillo. Y en casos más básicos como el de las bacterias, simplemente se basan en la ejecución de una o dos acciones (mover y multiplicarse).

3.4. Interfaz gráfica de usuario (GUI)

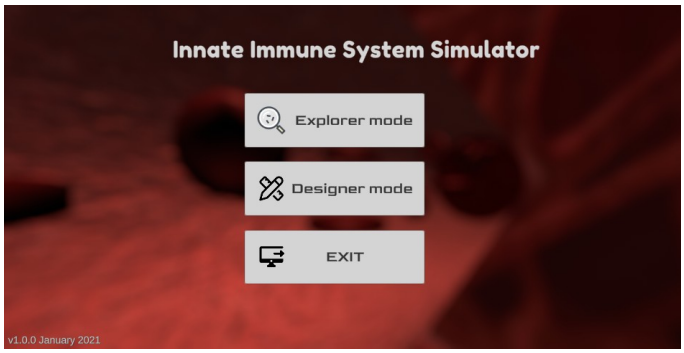


Figura 16: Pantalla principal del simulador

La interfaz gráfica del usuario está diseñada para que, en general, la experiencia no se vea saturada por una gran cantidad de información ni muchos elementos interactivos.

En cuestión de diseño gráfico, se buscaba un balance minimalista con menús simples (ver figura 16) que nos aportara la facilidad de que si un día se quiere adaptar el simulador a Android, la cantidad de elementos mostrada no sería un problema para dispositivos con pantallas mucho más reducidas.

3.4.1 Elementos gráficos y lógica

Para agilizar el desarrollo del proyecto se utilizaron algunos assets gratuitos (Lean) como base para que dieran forma a las ventanas y otros elementos.

Estos assets son elementos aislados casi puramente gráficos, por lo que una gran parte del trabajo dedicado en este apartado se centra en la lógica de gestión de ventanas, instanciación y manipulación.

La lógica de gestión de ventanas es fundamental para poder mostrar información al usuario de forma dinámica a través de código y es particularmente importante para el uso del modo Diseño.

El desarrollo de esta parte del proyecto consiste en lo siguiente:

1. La lógica empieza con un script que se encarga de gestionar todos los elementos gráficos que tengan que ver con los assets de Lean (LeanWindowManager)

- Este script requiere referencias a los prefabs para construir los elementos gráficos y el objeto Canvas sobre el cual se irán posicionando.
- Para simplificarlo lo máximo posible, se crearon funciones que con una sola llamada gestionan todo el proceso.
  - ShowInfo(...): muestra una ventana con la información indicada por argumento.
  - ShowQuestion(...): muestra una ventana de pregunta al usuario con hasta 3 posibles respuestas (figura 17). En cuanto el usuario hace click en OK, la ventana se cierra y muestra una nueva ventana de información (corrección) indicándole si su respuesta fue acertada.
  - ShowCorrection(…): muestra una ventana de corrección donde se compara la respuesta del usuario con la esperada y dibuja ciertos elementos de forma apropiada a si la respuesta fue acertada o no (fFiguraigura 18).
  - ShowHoverTooltip(…): crea un elemento de información en pantalla que solo muestra el texto cuando se pasa el cursor por encima.

2. Cada prefab que represente un elemento de Lean tiene su propio script *Handler.*
   - El script handler tiene como finalidad posibilitar la modificación de los elementos necesarios que posibilitan una instanciación dinámica (título, mensaje, etc.).
   - Este script es fundamental para que el LeanWindowManager acceda a las propiedades modificables del elemento que se quiere instanciar.

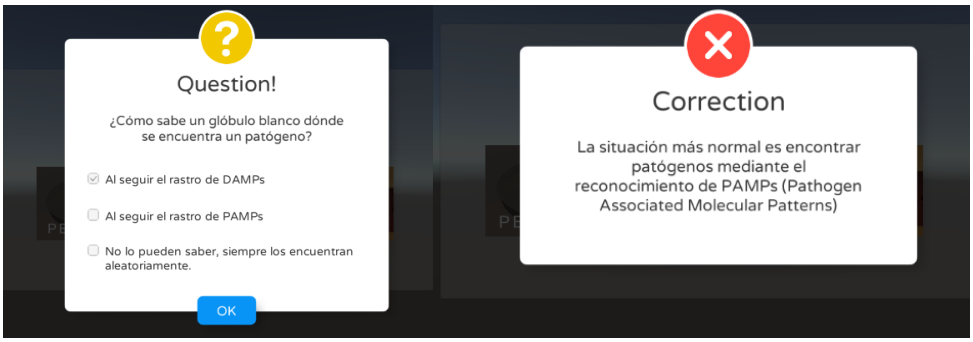


Figura 17: Ejemplo de pregunta (modificación de Lean)

Figura 18: Ejemplo de corrección (usando Lean)

Capítulo 4. Presupuesto de desarrollo.

Como se trata de un proyecto realizado en Unity cuya distribución está planificada para ser gratuita, no hay necesidad de pagar licencias ni por acceder a la propia aplicación en sí.

Varios de los elementos gráficos utilizados son libres y gratuitos y, en el caso de la mayoría de modelos 3D, fueron elaborados desde cero.

4.1 Resumen de gasto (temporal y económico).

A pesar de que los elementos externos y el motor utilizado no sean de pago, podemos contabilizar el tiempo de desarrollo como el gasto principal en este proyecto:

- **Fecha de inicio del proyecto:** Junio de 2020
- **Fecha de fin de proyecto:** Febrero de 2021
- **Semanas de trabajo:** 9 meses x 3 semanas c/u = 27 semanas
- **Horas trabajadas:** 27 semanas x 5 h c/u = 135 horas
- **Coste total:** 135 h x 25€/h = 3.375€

Capítulo 5. Conclusiones y líneas futuras

A raíz del desarrollo de este proyecto, se extrae una serie de conclusiones que involucran tanto apartados del mundo de la ingeniería como el de la biología. Aunque dado que este trabajo pertenece a la rama de ingeniería, este apartado se centrará mayormente en esta rama de la ciencia.

Primero, es interesante remarcar el hecho de que, a pesar de que este Trabajo Fin de Máster se hizo para un plan de estudios centrado en los videojuegos, podemos demostrar cómo es posible utilizar los conocimientos adquiridos para dar un enfoque distinto a las aplicaciones que se pueden crear. En este caso, no se ha elaborado un juego, se crea un simulador que permite acercar conocimientos de biología desde otra perspectiva utilizando el motor de Unity.

Entre los conocimientos adquiridos durante el Máster de Desarrollo de Videojuegos, fueron de especial interés para este proyecto:

1. Uso de Blender para la creación de modelos 3D y su importación en Unity (Motores y Sistemas de Desarrollo para Videojuegos).
2. Uso del entorno de Unity Engine y desarrollo de scripts en C# (varias asignaturas implicadas).
3. Optimización de recursos y código (Programación Optimizada para Videojuegos).
4. Inteligencia Artificial en Videojuegos.

Segundo, si bien la temática utilizada en este proyecto es bastante específica y muy alejada del mundo de los videojuegos, este proyecto ha servido conceptualmente para apreciar cómo la transversalidad de la informática es capaz de mejorar la didáctica de casi cualquier otra materia y hacernos ver que no todo está hecho ya o la escasez de este tipo de contenido.

Tercero, este proyecto se centró exclusivamente en algunos conceptos básicos del mundo de la inmunología, pero no por ello está cerrado a ser ampliado. Como se ha indicado en varios apartados, hay muchas situaciones y procesos que, dependiendo del nivel de profundidad que se quiere alcanzar, requiere de desarrollos aún mayores. Podría ser interesante tener varias ramas de este proyecto en las que se aproveche el desarrollo inicial como base para profundizar en distintos apartados según las necesidades de cada uno.

El código del proyecto quedará a disposición del público en el siguiente repositorio: https://github.com/miguelgdx/tfm_bio

Summary and Conclusions

As a result of this project development, we can draw a series of conclusions that involve topics from the world of Computer Science Engineering as well as Biology. However, given the fact that this work belongs to the Computer Science branch, this chapter shall be focused in that science field.

First, it is interesting to underline the fact that, despite this final master project was made in a study plan centered in videogames, we can demonstrate how it is possible to utilize the acquired knowledge to aim for a different goal in the applications that could be made. In this case, no videogame was made, but a simulator that allows the user to approach Biology knowledge from a different perspective by using the Unity engine.

Among the acquired knowledge throughout the *Videogame Development Master*, it was of special interest for this project:

1. Use of Blender for the making of 3D models and its importing in Unity (subject *Motores y Sistemas de Desarrollo para Videojuegos*).

2. Use of the Unity Engine environment and script development in C# (multiple subjects involved).

3. Resources and code optimization (subject *Programación Optimizada para Videojuegos*).

4. Artificial Intelligence in Videogames.

Second, even though this project topic is quite specific and very well far from the world of video games, this work makes it possible to appreciate how the transversality of computer science is able to improve the didactic of almost any subject or topic and allow us to notice how many things are yet to be made or the huge lack of content there might be.

Third, this project is focused exclusively in some basic aspects of the immunology world, but that does not mean it is closed to be extended. As stated in multiple chapters, there are many situations and processes that, depending on the level of depth that is intended to reach, it may require greater development. It might be interesting to have multiple branches of this project where they all leverage on the base just to portray different aspects or situations that may fulfill the needs of other audiences.

This project code shall be available for anyone in the following repository: https://github.com/miguelgdx/tfm_bio

Bibliografía

[1]: Institute for Quality and Efficiency in Health Care (IQWiG), The innate and adaptive immune systems, 2006, https://www.ncbi.nlm.nih.gov/books/NBK279396/#_i2255_theadaptiveimmunesys_

[2]: Gobierno de Canarias., Currículos de las materias y los ámbitos de la Educación Secundaria Obligatoria., 2016, https://www.gobiernodecanarias.org/cmsweb/export/sites/educacion/web/_galerias/descargas/bachillerato/curriculo/nuevo_curriculo/nuevas_julio_2015/troncales/03_biologia_geologia.pdf

[3]: Gobierno de Canarias., Currículo de las materias troncales y específicas para la etapa de Bachillerato, 2016, https://www.gobiernodecanarias.org/cmsweb/export/sites/educacion/web/_galerias/descargas/bachillerato/curriculo/nuevo_curriculo/nuevas_julio_2015/troncales/02_biologia.pdf

[4]: Inspark, Into the Cell, , 1. https://components.smartsparrow.com/components/into-the-cell/

[5]: Zane Hedges, P.tree: Single-Cell Simulator, , 2. https://chemistrychrist.itch.io/ptree-single-cell-simulator

[6]: Syndaus Inc., Immune Quest, , http://immunequest.com/

[7]: , Panda Behaviour, , http://www.pandabehaviour.com/

Anexos

WBC.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Panda;
public class WBC : MonoBehaviour
{
    protected PhysicsController pc;                     // Physics control scripts
    public List<Vector3> RecognizedPatterns;            // List of recognized patterns positions.
    private const int LIMIT_RECOGNIZED_PATTERNS = 5;    // Limit of recognized patterns
    protected float LastPatternCollision;               // Time when the last pattern collided
    private const float PATTERN_CLEAR_TIME = 3;         // Time to clear the whole list of
pattern recognition
    private const float DEACTIVATION_TIME = 4;          // Time that it takes the WBC to stop
being in active state
    private State CurrentState;                         // State of the current WBC
    protected bool Activated;                           // Is the WBC activated (as in Biology
meanings)
    public GameObject ContactedObject;                  // When other objects touch this WBC, it
shall get stored here.

    // Start is called before the first frame update
    protected void Start()
    {
        pc = GetComponent<PhysicsController>();
        CurrentState = new PatrolState(this);
        RecognizedPatterns = new List<Vector3>(LIMIT_RECOGNIZED_PATTERNS);
        // Depending on the WBC state and pattern list, execute a fixed behaviour.
        InvokeRepeating("UpdatePatterns", 2, 0.5f);
        Activated = false;
    }

    // Update is called once per frame
    protected void Update()
    {
        // If it's been enough time between hits, clear the whole list.
        if (Time.time - LastPatternCollision >= PATTERN_CLEAR_TIME)
        {
            pc.ClearInternalForces();
        }

        // Deactivation time does not necessarily have to match pattern clearn time.
        if (Time.time - LastPatternCollision >= DEACTIVATION_TIME)
        {
            Activated = false;
        }
    }

    // This function will be called by the DAMP object particle system if they collide with a
WBC.
    public void DAMPCollision(Vector3 point)
```

```
    {
        // Attraction to DAMPs?
        Activated = true;
        Vector3 DAMPReactionDir = (point - transform.position).normalized;
        AddPattern(DAMPReactionDir);
    }

    // This function will be called by the PAMP object particle system if they collide with a
WBC.
    // Note: Functions are split as it's unclear whether PAMPs and DAMPs have the same intensity
effect on a WBC.
    // This should be modified in the future if there is enough evidence of distinction between
these patterns.
    public void PAMPCollision(Vector3 point)
    {
        // Attraction to PAMPs?
        Activated = true;
        Vector3 PAMPReactionDir = (point - transform.position).normalized;
        AddPattern(PAMPReactionDir);
    }

    // Add the pattern to the list of contacted patterns.
    private void AddPattern(Vector3 p)
    {
        LastPatternCollision = Time.time;
        if (RecognizedPatterns.Count == LIMIT_RECOGNIZED_PATTERNS)
            RecognizedPatterns.RemoveAt(0);
        RecognizedPatterns.Add(p);
    }

    private void UpdatePatterns()
    {
        CurrentState.UpdateInternalForce(RecognizedPatterns);
    }
    [Task]
    public bool IsActivated()
    {
        return Activated;
    }
    [Task]
    public void FollowPatterns()
    {
        Debug.Log("Follow patterns!");
        Task.current.Succeed();
    }

    public virtual void HandleDebris()
    {
        Debug.Log("Handling debris!");
    }

    [Task]
    public bool FoundPathogen()
    {
        if (ContactedObject != null && ContactedObject.tag == "Pathogen")
            return true;
```

```
    return false;
}

[Task]
public bool FoundDebris()
{
    if (ContactedObject != null && ContactedObject.tag == "DamagedCell")
        return true;

    return false;
}

[Task]
public void Engulf()
{
    // TODO: Check if bacteria can be engulfed.
    if (FoundPathogen())
    {
        RemovePathogen(ContactedObject);
        Task.current.Succeed();
    }
    Task.current.Fail();
}

protected void RemovePathogen(GameObject obj)
{
    GameObject.Destroy(obj);
}

protected void RemoveDebris(GameObject obj)
{
    GameObject.Destroy(obj);
}

[Task]
public void SignalInflammation()
{
    Task.current.Fail();
}

public void Degranulate()
{

}


public abstract class State
{
    protected WBC InstanceReference;
    public State(WBC reference)
    {
        InstanceReference = reference;
    }
    // Every few seconds, the neutrophil will update it's internal force.
    public abstract void UpdateInternalForce(List<Vector3> RecognizedPatterns);
```

```
    }

    public class PatrolState : State
    {
        public PatrolState(WBC reference) : base(reference)
        {
        }

        public override void UpdateInternalForce(List<Vector3> RecognizedPatterns)
        {
            Vector3 finalDir = Vector3.zero;
            if (RecognizedPatterns.Count == 0)
                return;

            finalDir = RecognizedPatterns[RecognizedPatterns.Count - 1];
            finalDir = finalDir.normalized;

            // Tell the physics controller that there is an internal force (attraction) going.
            PhysicsController.InternalForce InternalForce = new
PhysicsController.InternalForce(finalDir, (float)RecognizedPatterns.Count * 0.5f);
            InstanceReference.pc.ApplyInternalForce(InternalForce);
        }
    }

    public class PassiveState : State
    {
        public PassiveState(WBC reference) : base(reference)
        {
        }

        public override void UpdateInternalForce(List<Vector3> RecognizedPatterns)
        {
            Vector3 finalDir = Vector3.zero;
            if (RecognizedPatterns.Count == 0)
                return;

            // Perform a vector sum of all pattern positions and then just normalize it.
            for (int i = 0; i < RecognizedPatterns.Count; i++)
            {
                finalDir += RecognizedPatterns[i];
            }
            finalDir = finalDir.normalized;

            // Tell the physics controller that there is an internal force (attraction) going.
            PhysicsController.InternalForce InternalForce = new
PhysicsController.InternalForce(finalDir, (float)RecognizedPatterns.Count * 0.5f);
            InstanceReference.pc.ApplyInternalForce(InternalForce);
        }
    }

}
```

PhysicsController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PhysicsController : MonoBehaviour
{
    private Rigidbody _rb;
    public List<ExternalForce> ExternalForces;
    public List<InternalForce> InternalForces;
    public Vector3 _resultDir;
    public float _resultMagnitude;
    public float currentSpeed;
    public float countExternalForces;
    public bool RandomTorque = true;
    private Collider col;
    public bool ExternalForcesEnabled = true;
    public bool InternalForcesEnabled = true;

    // Start is called before the first frame update
    void Awake()
    {
        ExternalForces = new List<ExternalForce>();
        _rb = GetComponent<Rigidbody>();
        col = GetComponent<Collider>();
        InternalForces = new List<InternalForce>();
    }

    private void Start()
    {
        // In case we want the model to rotate randomly to make it look "cooler"
        Vector3 RotDir = Vector3.zero;
        if (RandomTorque)
        {
            RotDir = new Vector3(
                Random.Range(-1f, 1f),
                Random.Range(-1f, 1f),
                Random.Range(-1f, 1f));
            _rb.AddTorque(RotDir * 10f);
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        // If the object hits a clotting top, stop its velocity.
        if (other.gameObject.tag.Equals("ClotTop"))
        {
            _rb.constraints = RigidbodyConstraints.FreezePositionY;
        }
    }

    // Update is called once per frame
```

```
    void Update()
    {
        countExternalForces = ExternalForces.Count;
        if (ExternalForcesEnabled == true)
        {
            // Loop over external forces and obtain the resultant direction.
            Vector3 totalForce = Vector3.zero;
            foreach(ExternalForce ef in ExternalForces)
            {
                Vector3 curForce = ef.Dir * ef.Intensity;
                totalForce = totalForce + curForce;
            }
            // Now we can have the unitary vector and magnitude.
            _resultMagnitude = totalForce.magnitude;
            _resultDir = totalForce.normalized;

            Vector3 forceDifference = (_resultDir * _resultMagnitude) - _rb.velocity; // V = Vf -
V0

            // Apply force to rb
            _rb.AddForce(forceDifference);
            Debug.DrawLine(this.transform.position, transform.position + forceDifference);
            currentSpeed = _rb.velocity.magnitude;

        }

        if (InternalForcesEnabled == true)
        {
            // Apply Internal forces.
            foreach (InternalForce inf in InternalForces)
            {
                _rb.AddForce(inf.Dir * inf.Intensity);
                Debug.DrawRay(transform.position, inf.Dir, Color.magenta);
                Debug.Log("Internal force: " + inf.Dir);
                Debug.Log("current speed: " + _rb.velocity);
            }
        }
    }

    public void applyExternalForce(Vector3 dir, float intensity, GameObject origin)
    {
        if (ExternalForces.Count > 0) {
            // Check if the origin already existed in our list of applied forces.
            foreach (ExternalForce ex in ExternalForces)
            {
                if (ex.Origin == origin)
                    return;
            }
        }

        ExternalForce ef = new ExternalForce(dir, intensity, origin);
        ExternalForces.Add(ef);
    }

    public void RemoveExternalForce(GameObject origin)
    {
        // Remove the object if it matches anything on the list.
```

```
        foreach (ExternalForce e in ExternalForces)
            if (e.Origin == origin)
            {
                ExternalForces.Remove(e);
                return;
            }
    }

    // A single immediate push force (just once).
    public void Push(Vector3 dir)
    {
        _rb.AddForce(dir, ForceMode.Impulse);
    }

    public void ConstantPush(Vector3 dir)
    {
        _rb.AddForce(dir, ForceMode.Force);
    }

    /**
     * Class to define each of the external forces that are being applied onto the affected
objected.
     */
    public class ExternalForce
    {
        private Vector3 _dir;
        private float _intensity;
        private GameObject _origin;

        public ExternalForce(Vector3 Dir, float Intensity, GameObject Origin)
        {
            this.Dir = Dir;
            this.Intensity = Intensity;
            this.Origin = Origin;
        }
        public Vector3 Dir
        {
            get { return _dir; }
            set { _dir = value; }
        }
        public float Intensity
        {
            get { return _intensity; }
            set { _intensity = value; }
        }
        public GameObject Origin
        {
            get { return _origin; }
            set { _origin = value; }
        }
    }

    /**
     * For now, InternalForces may be treated the same as ExternalForces, but it would
     * be better for future works to make a distinction on how forces really work from
     * inside one's body
```

```
     */
    public class InternalForce : ExternalForce
    {
        public InternalForce(Vector3 Dir, float Intensity) : base(Dir, Intensity, null)
        {
        }
    }

    public void ApplyInternalForce(InternalForce force)
    {
        InternalForces.Clear();
        InternalForces.Add(force);
    }

    public void ClearInternalForces()
    {
        InternalForces.Clear();
    }

    // Freezes the rigidbody
    public void Freeze()
    {
        _rb.constraints = RigidbodyConstraints.FreezeRotationX |
RigidbodyConstraints.FreezeRotationZ | RigidbodyConstraints.FreezePositionY |
RigidbodyConstraints.FreezePositionZ;
    }

    // Unfreezes the rigidbody, allowing it to move again
    public void Unfreeze()
    {
        _rb.constraints = RigidbodyConstraints.None;
    }
}
```