|  |
| --- |
| **1. Introducción** |

El objetivo de la presente instancia es evaluar el progreso del alumno en la comprensión de los temas tratados en las unidades abordadas por la materia. Esta evaluación constará de una sección teórica y una sección práctica. El alumno dispondrá de siete días para realizar la parte práctica y deberá defenderla oralmente junto con la evaluación teórica.

|  |
| --- |
| **1.1 Criterios de Aprobación** |

Para considerar a esta evaluación como aprobada, el alumno deberá aprobar de forma **individual** tanto la parte teórica como la parte práctica. Se entiende que aprobó la sección teórica cuando contestó correctamente el 60% de las preguntas orales. Se entiende que aprobó la sección práctica cuando el alumno puede no sólo ejecutar la build cumpliendo con todas las consignas enunciadas sino también explicar el código fuente con sus palabras.

|  |
| --- |
| **2. Evaluación Teórica** |

|  |
| --- |
| **Las preguntas serán realizadas por el docente en forma sincrónica mediante videoconferencia.** |
| Porcentaje de preguntas aprobadas: |

|  |
| --- |
| **3. Evaluación Práctica** |

El objetivo del presente ejercicio es evaluar las capacidades de uso de un motor gráfico, la aplicación de los conceptos teóricos, y la implementación de los recursos prácticos abordados por la asignatura en las unidades incluidas en este examen.

|  |
| --- |
| **3.1. Restricciones** |

* Se requiere que el alumno realice una build de un videojuego con Unity 2019.4.12 (Universal Render Pipeline) y C#.
* Para considerar a una consigna cumplida, la funcionalidad planteada por la misma debe poderse probar en su totalidad.
* El parcial deberá subirse también a un repositorio público así el docente puede ver la historia de los commits.
* No se evaluará la estética del prototipo. Sin embargo, se pide utilizar materiales con colores distintos para hacer más claro el diseño.
* Se deberá subir 3 archivos a la carpeta compartida que indicaron en el formulario con permiso de lectura para cualquier persona. El primero será un video mp4 no mayor a dos minutos que muestre las consignas en funcionamiento y el código desde el editor. El segundo será un .zip con el código fuente. Borrar la carpeta Library del código fuente así pesa menos MB. El tercero será un .txt con un link al repositorio público del proyecto.

El nombre de cada archivo deberá respetar el siguiente formato: Apellido\_Nombre\_1ParcialSimuladores202X\_TurnoX.zip y Apellido\_Nombre\_1ParcialSimuladores202X\_TurnoX.mp4.

Apellido\_Nombre\_1ParcialSimuladores\_REPOSITORIO\_202X\_TurnoX.txt.

* El juego no debe mostrar errores por consola.
* NO usar colliders/OnTiggerEnter/OnCollisionEnter para detectar las colisiones en el presente juego (a excepción de la sexta consigna).

|  |
| --- |
| **3.2. Recomendaciones** |

Se recomienda al alumno leer con detenimiento las restricciones y consignas enunciadas.

|  |
| --- |
| **3.3. Contextualización** |

El Sokoban es un juego de puzzle tradicional japonés. El objetivo es depositar todos los bloques en los casilleros verdes para pasar de nivel.

Jugador se mueve de a un casillero (izquierda, derecha, arriba, abajo). Jugador puede empujar un bloque de a un casillero hacia adelante sí y solo sí el lugar destino está libre. Es decir, si no hay pared, otro bloque o fin del tablero.

Prefabs utilizados: Bloque, Casillero, CasilleroTarget, Jugador, Pared.

La clase Nivel representa un nivel. Posee nombre y tablero.

La clase Tablero representa un tablero. Permite agregar/modificar/consultar a sus objetos (Casillero, Casillero Target, Bloque, Pared, Jugador). También devuelve las posiciones de dichos objetos como Vector2 o lista de Vector2.

La clase InstanciadorPrefabs permite instanciar los prefabs según el tablero actual para que Unity los grafique. Como es un juego por turnos se llama al inicio y por cada movida.

La clase SokobanLevelManager recibe los prefabs desde el Inspector y devuelve un nivel con un tablero particular.

La clase SokobanGameManager controla el flujo de ejecución del juego. Inicializa el juego y lee las acciones del usuario. Implementa las mecánicas de juego. Informa cuándo gana el jugador.

|  |
| --- |
| **3.4. Consignas** |

|  |  |  |
| --- | --- | --- |
| 1 | 1. Hacer que el Jugador pueda moverse de a un casillero hacia abajo y a la izquierda (teclas S/A y flechas abajo/izquierda). |  |
| 2 | Hacer que el Jugador no pueda avanzar cuando intente mover dos o más bloques seguidos. |  |
| 3 | Hacer que se detenga el juego cuando el jugador deposita todos los bloques en los casilleros target. La lógica para chequear la condición de victoria debe contemplar que los casilleros target pueden estar en cualquier lugar del tablero. No debe “hardcodearse”. Imprimir un mensaje en consola para informar que ganó. |  |
| 4 | 1. Hacer que se pueda deshacer movidas presionando la tecla Z. Esto implica revertir las posiciones del jugador y bloques desde el último movimiento hasta el inicio del nivel. Aclaración: si un jugador se mueve 100 veces contra una pared que lo bloquea, no debería pulsar 100 veces "z" para deshacer esa movida ya que, si bien hubo intención de movimiento, el mismo no se efectuó. 2. Tip: usar una pila (Stack). |  |
| 5 | 1. Generar el nivel en base a una imagen compuesta por pixeles. El color de cada píxel representará un prefab distinto. |  |
| 6 | 1. Elaborar mecánicas que hagan que la experiencia de juego sea más intensa. Ej.: proyectiles que se disparen por filas/columnas y haya que esquivarlos. Baldosas vacías que desaparezcan. Etc. Agregar un tiempo límite para solucionar cada nivel. Mostrarlo en la UI. Se pueden usar colliders y triggers para esta consigna. |  |