

TECHNICAL DESIGN DOCUMENT

CITY BUILDER

Motores del Videojuego II
Iván Sancho
Curso 2020/2021



|



Plantilla desarrollada para la asignatura
“Motores Gráficos II” de 2º del HND en
“Diseño de Videojuegos”



Índice

1. Introducción a la <i>template</i>	5
1.1. Funcionalidad entregada	5
1.1.1. Niveles de ejemplo	7
1.2. Sistemas de la plantilla	7
1.2.1. Sistema de navegación	7
1.2.2. Sistema de recursos	8
1.2.3. Sistema de construcción	9
1.2.4. Sistema de tecnologías	10
1.3. Funcionalidad a desarrollar	10
2. Panorámica visual	11
3. Clases y herencias	19
3.1. Jerarquía de herencias	19
3.1.1. <i>Framework</i> principal	19
3.1.2. Edificios	20
3.1.3. Recursos	21
3.1.4. NPC	21
3.1.5. Tareas del árbol de tecnologías	22
3.1.6. <i>Widgets</i>	22
3.1.7. Otras clases	23
3.2. Interfaces	24
3.3. Enumeraciones	24
3.4. Estructuras de datos	25
3.4.1. Taxonomía	27
3.5. Tablas de datos	28
4. <i>Flow</i> del input	29
5. Modificación de la plantilla	31
5.1. Configuración en edición de “BP_GridWorld”	31
5.2. Terminar de preparar el mundo de juego	33
5.3. Revisar el estado del EQS	35
5.4. Tener lista la “Navigation Mesh”	35
5.5. Añadir recursos de materias primas	36
5.6. Añadir recursos manufacturados	38
5.7. Generar recursos monetarios	38
5.8. Añadir nuevos tipos de edificios	39
5.8.1. Especificidades en edificios básicos	43
5.8.2. Especificidades en edificios recolectores	43



5.8.3. Especificidades en edificios productores	43
5.8.4. Especificidades en edificios almacenes	43
5.9. Instanciar edificios multicasillas	44
5.10. Instanciar edificios en zonas	44
5.11. Edificios y “Navigation Mesh”	45
5.12. Interacción entre sí de los agentes de IA	45
5.13. Configurar año, edad y velocidad <i>in game</i>	47
5.14. Misiones de un nivel	47
5.15. Campañas de varios niveles	47
5.16. Configurar el árbol de tecnologías	47
5.17. Interfaz de construcción	52
5.18. Edificios únicos	55
5.19. Door point	57
5.20. Impedir instanciar un edificio	57
5.21. Cambiar el nombre a la carpeta base	57
5.22. Escalado de una geometría y <i>zoom</i> dentro de ella	58
5.23. Modificación de las estructuras de datos	59
5.24. Consideraciones finales	61



1. Introducción a la template

El objetivo del presente documento es ofrecer una panorámica general de funcionalidad desarrollada en la plantilla entregada al diseñador con el fin de poder realizar un proyecto del género “City Builder”. Es importante no confundir el género a desarrollar con otros similares del estilo RTS.

El objetivo de la práctica es doble: por una parte, poner en acción los conocimientos de *scripting* adquiridos previamente en un motor con capacidad AAA como es “Unreal Engine 4” con el fin de desarrollar funcionalidad propia; y, por otra parte, iterar y trabajar el balanceo del juego y sus recursos en base a los conocimientos aprendidos en asignaturas complementarias.

En este documento se va a abordar la problemática de la relación de datos y funcionalidad programada, haciendo hincapié en aquellos puntos que puedan resultar de especial interés al diseñador para modificar ciertas partes de funcionalidad y adaptarlas a su diseño de juego. Así pues, se va a poner el foco acerca de qué partes de la funcionalidad le son liberadas al diseñador junto con la plantilla inicial, se va a ofrecer una panorámica general de la funcionalidad desarrollada y se va a resumir a grandes rasgos la funcionalidad común a programar por el diseñador.

1.1. Funcionalidad entregada

A grandes rasgos, la plantilla contiene:

- Una serie de clases, no todo actores, que conforman el *framework* principal de clases de la plantilla asociadas con un modo de juego por defecto: “BP_GameModeBase”, “BP_GameState”, “BP_HUD”, “BP_Pawn”, “BP_PlayerController”, “BP_PlayerCameraManager” y “BP_GameInstance”.
- La clase “BP_Pawn” es una clase que hereda directamente de “pawn” y es la que ejerce de personaje principal por defecto de la escena. El personaje principal es un personaje implícito puesto que no se va a ver nunca como tal sino más bien mediante alguno de sus componentes como por ejemplo el cursor del decal para visualizar hacia dónde está apuntando el jugador en cada momento. No es un “character” porque no hay necesidad de emplear sus componentes.
- La clase “BP_PlayerController” canaliza el input mapeado en “Project Settings” hacia la clase “BP_Pawn” principalmente. Hay algún input relacionado con un eventual menú de pausa especialmente preparado para ser canalizado hacia la clase “BP_HUD”, pero esa funcionalidad no ha sido todavía desarrollada y se deja a decisión del diseñador.
- Dentro del “BP_PlayerController”, además, se encuentra asociada la clase “BP_PlayerCameraManager”, de momento sin cambios sustanciales, pero creada por si el diseñador tuviera que realizar alguna acción o configuración específica relacionada con la gestión de cámaras.
- La clase “BP_Pawn” cobra especial importancia porque es en la que se gestiona todo lo referente con la navegación por el mundo de juego y la instanciación de edificios. La información que en ocasiones pueda necesitar la va a tomar de una instancia que ha de haber sí o sí en el mundo de la clase “BP_GridWorld”.
- La clase “BP_GridWorld” es, junto con la clase “BP_Pawn”, la otra clase esencial para que el sistema de navegabilidad funcione ya que la información que se almacena o calcula en esta clase permite, por ejemplo, dar a conocer al actor poseído por el jugador si se encuentra en un escenario de mundo en *grid* cuadrangular o más bien un mundo libre. Esta clase, además, funcionará en base a un par de objetos de la clase “BP_Beacon” instanciados en el mundo en dos esquinas antagónicas y que permitirán conocer sus límites.



- En cambio, en esta ocasión, la clase “BP_GameInstance” carece de gran protagonismo puesto que no hay nada que, de antemano, se haya considerado que tenga que persistir de un nivel a otro: en el momento de la entrega de la plantilla: lo único que hay son declaraciones de eventos.
- Por otra parte, es en la clase “BP_GameState” donde se almacenará la información del juego relacionada con los recursos de que dispondrá el jugador, que inicialmente son: monedas, madera, hierro y oro. El diseñador puede modificarlos a su consideración, añadiendo o eliminando recursos según su idea de juego, pero ha de hacerlo dentro de esta clase y ha de modificar la funcionalidad relacionada en consecuencia.
- Además, es en esta clase donde se almacenará también otros datos referentes al estado de la partida como la población, trabajadores ocupados, edad, mes y año de dentro del juego. También incluye información sobre las tecnologías descubiertas y los edificios únicos que han sido construidos.
- En la clase “BP_HUD” se crea el principal *widget* de la escena, *widget* de nombre “WB_HUD”. Es en este *widget* en el que se incluyen ya de antemano otros *widgets* secundarios como la barra superior de estado general de la partida o las páginas de construcción de nuevos edificios, siguiendo un sistema modular en el que se puedan crear tantas opciones de diseño de interfaz como sean necesarias.
- Adicionalmente, la plantilla incluye una serie de clases base para heredar de ellas y crear nuevos tipos de edificio y desarrollar su funcionalidad: “BP_BasicBuilding”, “BP_CollectorBase”, “BP_ProducerBase” y “BP_StorageableBase”. La clase especial “BP_PrefabBuilding” es una clase que está instanciada una única vez por defecto, de la que preferiblemente no se heredará nunca y de la que tampoco se instanciarán más actores, ni en tiempo de edición ni en tiempo de ejecución. Estas clases básicas de las que heredar se encuentran en la jerarquía de carpetas “/Content/StudentName/Placeables/Buildings” y siguientes.
- Asimismo, también hay una clase de nombre “BP_BasicResource” de la que heredaremos cuando tengamos necesidad de crear nuevos recursos de materias primas para instanciarlos por el nivel. Esta clase prefabricada se encuentra dentro de la jerarquía de carpetas “/Content/StudentName/Placeables/Resources”.
- A destacar la clase “BP_ZoneTrigger”, que se utilizará en diversas zonas del nivel de juego para determinar zonas en las que el jugador podrá instanciar determinados edificios.
- Los agentes de inteligencia artificial que han de ir de los edificios recolectores a las materias primas y de los edificios productores a los almacenes ya están programados de antemano mediante las clases “BP_Worker” y “BP_Transporter”, ambas derivadas de “BP_NPC” como clase base para agrupar atributos y funcionalidad, clase hija a su vez de la clase “ACharacter” del motor. Destacar que el diseñador puede realizar cualquier cambio que considere oportuno en el comportamiento de ambos agentes para adaptarlos a su idea de juego, aunque en principio no se atisba como necesario ni se recomienda realizar modificación alguna. A destacar que, aunque un edificio tenga asignados varios aldeanos como trabajadores, un único agente de inteligencia artificial saldrá de cada edificio, sólo que tal vez este represente a varios trabajadores o a ninguno.
- La plantilla incluye un árbol de tecnología para poder desbloquear nuevas tecnologías o edades, algo a plantear totalmente por parte del diseñador. Este árbol de tecnologías se basa en unos mini actores de tareas que heredarán de la clase “BP_TechTreeTask” y del que únicamente hay que reimplementar, en sucesivas herencias, el método “ExecuteTechTreeTask” para que por herencia y polimorfismo el diseñador programe la funcionalidad que sea en cada clase derivada.



1.1.1. Niveles de ejemplo

Junto con la plantilla se le entrega al diseñador un par de niveles de ejemplo: "TestMap1_P" y "TestMap2_P". Ambos se encuentran localizados dentro de la carpeta "/Content/StudentName/Maps" y en ambos prácticamente hay los mismos elementos: algunos edificios de cada tipo de prueba, algunos recursos, y una jugabilidad muy básica en la que hay aldeanos que van a por recursos o bien directamente a la materia prima o bien a un almacén.

La principal diferencia entre ambos niveles es que uno funciona sobre un plano y el otro demuestra funcionalidad sobre una *landscape* porque, por lo demás, ambos son exactamente iguales. En todo mapa nuevo que se cree es necesario tener instanciados por el nivel un objeto de las clases "BP_PrefabBuilding", "BP_Pawn" y "BP_GridWorld".

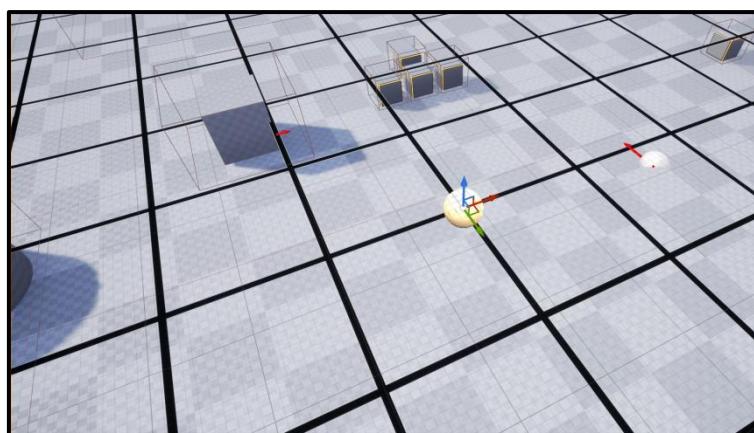
1.2. Sistemas de la plantilla

La plantilla incluye una serie de sistemas básicos y programados: navegación, recursos, construcción y tecnologías. En este subapartado se va a incidir a grandes rasgos en la naturaleza de estos sistemas principales para que el diseñador comprenda qué contiene el proyecto que le es entregado y comenzar a entender cómo trabajar con ello.

1.2.1. Sistema de navegación

El primer sistema a comentar de la plantilla es el sistema de navegación, el más básico y fundamental. Para que este sistema pueda funcionar, es imprescindible que el diseñador coloque sobre el mundo de juego, en tiempo de edición, un actor de las siguientes clases: "BP_Pawn" y "BP_GridWorld". El actor que instanciemos de la clase "BP_GridWorld" va a determinar:

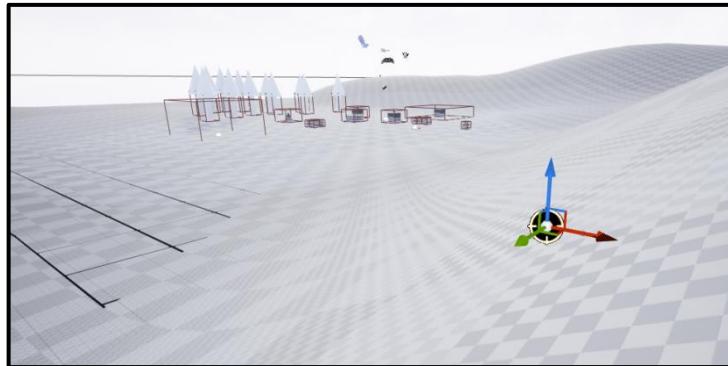
- Si el mundo de juego permite la instancia libre de edificios o en forma de *grid*.
- Si la forma del mundo de juego es rectangular o circular.
- Ángulo máximo en el que poder instanciar un edificio en tiempo de ejecución



Adicionalmente, es importante destacar que este actor funciona junto con un par de objetos de la clase "BP_Beacon" que hay que instanciar en ambas esquinas de juego para delimitar de forma clara los límites por exceso o defecto: uno tomará valores mínimos y el otro tomará valores máximos en base a sus respectivas posiciones en el mundo. En el caso de tratarse de un mundo de juego con forma circular y no cuadrangular, el sistema toma esos valores como radiales absolutos.



Así pues, se han de instanciar sendos actores de la clase “BP_Beacon” en las esquinas del mundo y asociarlos con el actor de la clase “BP_GridWorld” que también tengamos instanciado en el nivel. Son precisamente estos dos actores baliza, junto con los atributos de la *grid* que permiten definir el número de filas y de columnas del mundo de juego, los que van a determinar lo que ocupará una casilla en un mundo que tenga forma de *grid*.

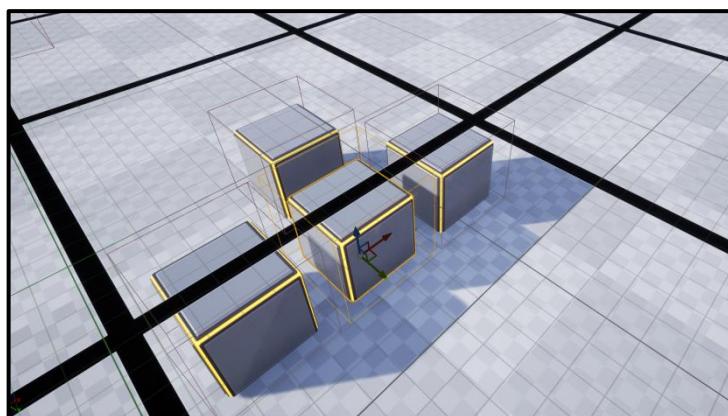


El actor de la clase “BP_Pawn” que posea el jugador va a ser el encargado de trabajar con la información que le pasa el actor de la clase “BP_GridWorld” para saber hasta qué punto puede navegar por el nivel y, por lo tanto, instanciar algún edificio en él. Todo lo relativo a navegación se encuentra en esta clase “pawn”: movimiento, gestión con los límites de la ventana gráfica y del terreno, zoom de la cámara, apuntado sobre el terreno con el cursor del ratón, rotación, selección de edificios para eliminarlos u obtener información sobre ellos, selección de recursos para obtener información sobre ellos, renderizado de la escena...

1.2.2. Sistema de recursos

La plantilla contiene una serie de recursos básicos o materias primas esenciales que el diseñador puede modificar a su antojo, por defecto: monedas, madera, hierro, oro. Es en tiempo de edición cuando el diseñador planteará, por una parte, los tipos de recursos que va a haber en su juego; y, por otra parte, su predisposición en el mundo de juego, cuántas unidades va a poseer el recurso, si se puede consumir, si ha de regenerarse y cuánto va a tardar en hacerlo...

Es en la clase base “BP_BasicResource” donde se encuentra ya desarrollada toda la funcionalidad relacionada con la gestión de las cantidades asociadas con el recurso así como los atributos y componentes básicos que todo recurso necesita, por lo que el diseñador se ha de preocupar únicamente de heredar de los recursos y darle a cada clase derivada la información que le corresponda.



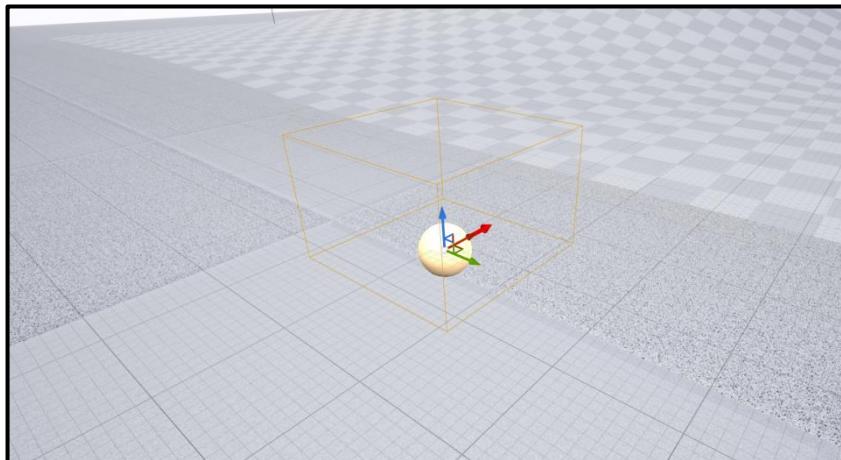


Por otra parte, el agente de inteligencia artificial “BP_Worker” será el encargado de ir a por el recurso y regresar a su edificio una vez que lo haya obtenido directamente de la materia prima, pero es funcionalidad que también le es entregada al desarrollador junto con la plantilla básica.

1.2.3. Sistema de construcción

Este sistema presenta una mayor complejidad que el anterior aunque también se basa en jerarquía de herencias: la plantilla incluye una clase base “BP_BasicBuilding” de la que heredan el resto de edificios.

Es importante destacar, en primer lugar, la clase “BP_PrefabBuilding” como clase especial que hereda directamente de “BP_BasicBuilding” y que ha de encontrarse previamente instanciada en el mundo de juego en tiempo de edición y que es la que permite a la clase “BP_Pawn” del jugador previsualizar en el nivel cómo se instanciará cada nuevo edificio, funcionalidad ya desarrollada por defecto.



La plantilla incluye dos clases derivadas de “BP_BasicBuilding” que son esenciales a la hora de crear la gran mayoría de edificaciones: “BP_CollectorBase” para aquellos edificios que tengan que mandar aldeanos a recolectar materias primas, y “BP_ProducerBase” para aquellas construcciones que tengan que, por el contrario, tengan que mandar a trabajadores a por recursos a un almacén. Ahí es donde interviene en escena la tercera y última clase derivada de “BP_BasicBuilding” de la que derivar para crear edificios core: “BP_StorageableBase” como clase base de la que heredarán todos los edificios de los que los aldeanos podrán recolectar recursos.

Remarcar que, en caso de que un edificio no sea ni productor, ni recolector, ni almacén, es perfectamente posible heredar directamente de la clase “BP_BasicBuilding” como se hace en la plantilla en el caso de, por ejemplo, con las casas –clase “BP_House”-. De esta forma, la plantilla ya presenta algunos edificios con funcionalidad propia a modo de demostración para el diseñador como son casas, un campamento maderero, una serrería, almacenes y un centro urbano –tratado en esta ocasión como un almacén–.

Finalmente, es imprescindible destacar que la plantilla ya está preparada para albergar edificios únicos y que, si un edificio único ha sido construido o un edificio cualquiera todavía no ha sido descubierto porque la tecnología asociada no ha sido investigada, este no saldrá como opción en los distintos menús de construcción. Además, si el edificio sólo se permite construir en una zona en concreta del nivel, el diseñador cuenta con la clase “BP_ZoneTrigger” para ir delimitando zonas por el mundo de juego.



Es la decisión del diseñador plantear los distintos tipos de edificios que va a haber en su juego, sus atributos específicos, si son únicos o mediante qué tecnología se van a descubrir, si se pueden construir sólo en una zona en concreto, así como programar su funcionalidad específica.

1.2.4. Sistema de tecnologías

La plantilla contiene un último sistema desarrollado relacionado con un árbol de tecnologías que permitirá al diseñador plantear mejoras tanto a nivel de descubrir edificios que construir como avanzar de época, así como determinar el nombre, descripción, coste y edad necesaria de cada una de las tecnologías que plantea.

Es un sistema que se basa en heredar de la clase actor “BP_TechTreeTask” para sobreescribir su método “ExecuteTechTreeTask” en sucesivas clases derivadas de forma que cada tecnología lo que hace es instanciar la correspondiente clase hija y ejecutar su funcionalidad mediante referencia a la clase padre por herencia y polimorfismo. El planteamiento de que esto se haya programado de esta forma no es por hacerlo lo más óptimo posible sino que se ha tomado la decisión de trabajarla mediante pequeños actores para comodidad del diseñador.

1.3. Funcionalidad a desarrollar

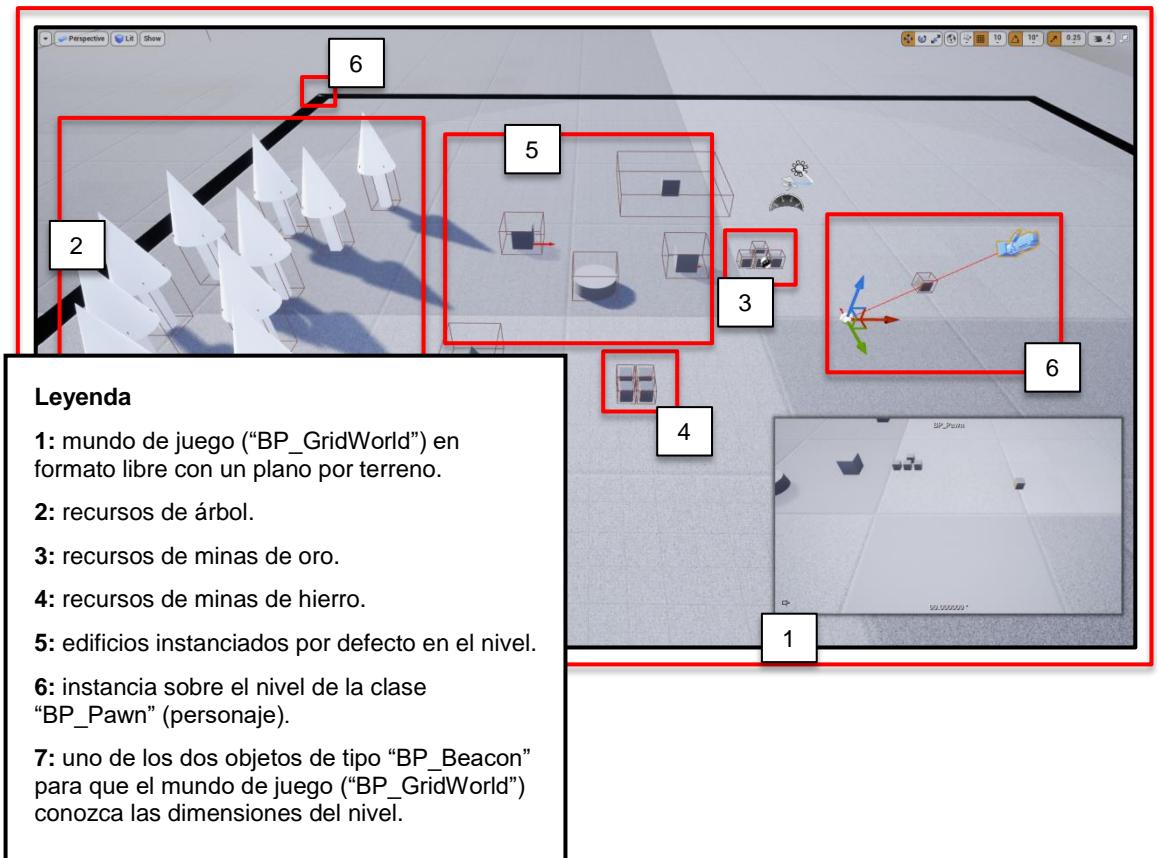
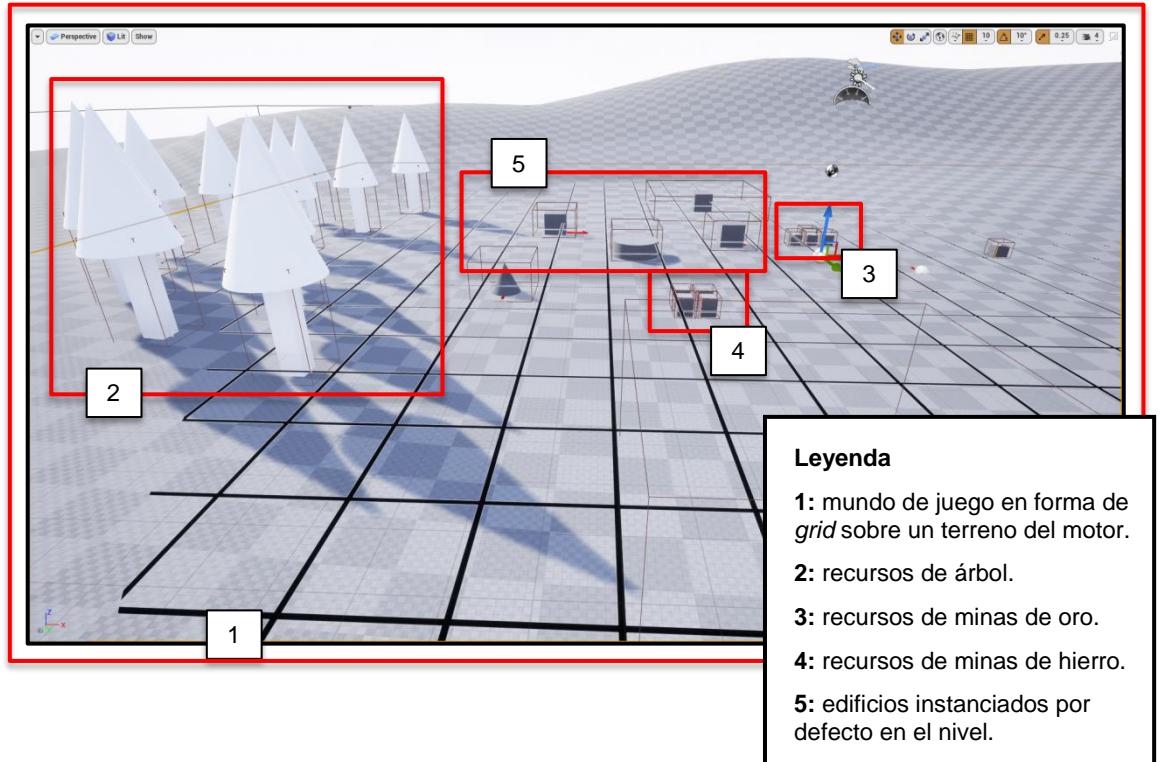
Una vez se ha ofrecido una panorámica general de lo que contiene la plantilla en el momento en el que esta le es liberada al diseñador, ¿qué es lo que este ha de desarrollar? ¿Qué opciones tiene en base al contenido previamente programado? El diseñador tiene libertad para decidir todo:

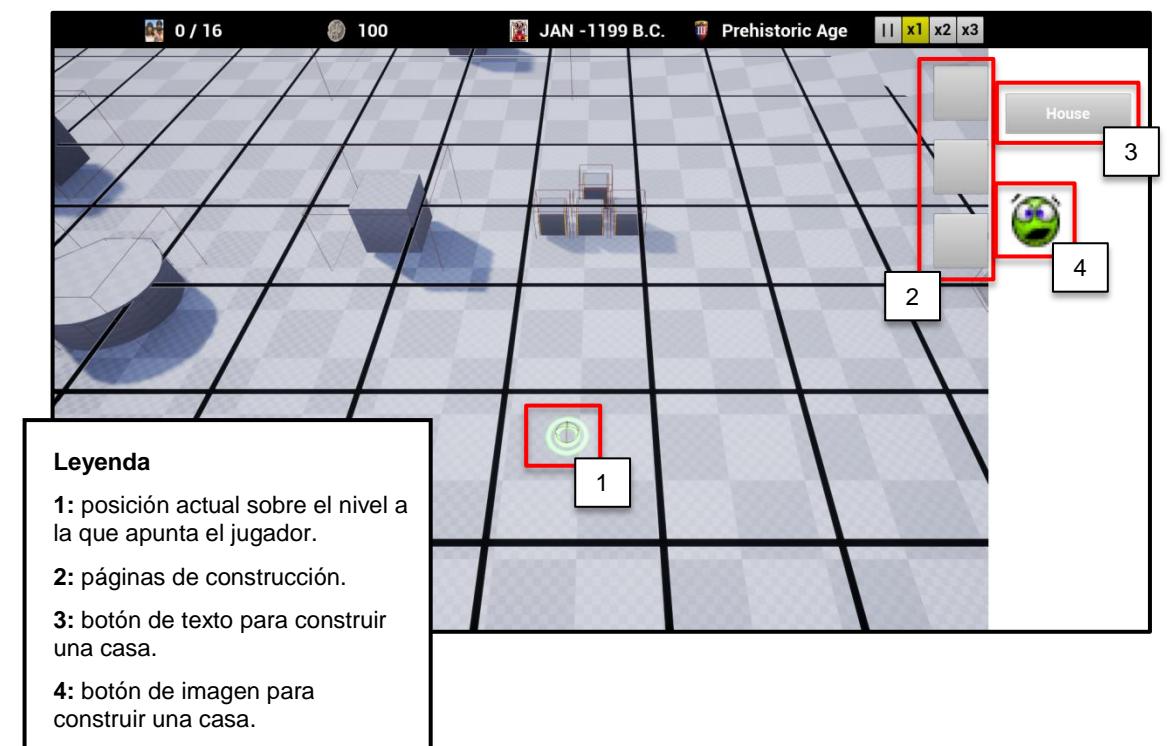
- Plantear la ambientación, época, cultura, mundo de juego, contexto narrativo y velocidad del juego.
- Decidir las dimensiones y forma del mundo de juego, teniendo la opción de emplear *landscapes* del propio motor.
- Determinar los recursos con los que va a poder jugar el usuario.
- Crear los distintos tipos de edificios que el jugador va a poder instanciar y desarrollar su funcionalidad.
- Desarrollar el árbol de tecnologías que va a poder explorar el jugador, así como crear y programar el contenido de las tareas a ejecutar por cada tecnología.
- Diseñar la interfaz de usuario de todos los elementos visibles por pantalla: barra superior, páginas de construcción, *widgets* de información de edificios y recursos, árbol de tecnologías...
- Plantear al menos un nivel jugable con una misión que se supere en el momento en el que el jugador consiga uno o varios de hitos relacionados con su ciudad: recolectar una serie de materias primas, almacenar un número concreto de recursos, descubrir la tecnología que sea, llegar a un número determinado de población... Adicionalmente, el diseñador puede plantear una campaña compuesta por, al menos, dos niveles distintos y consecutivos.

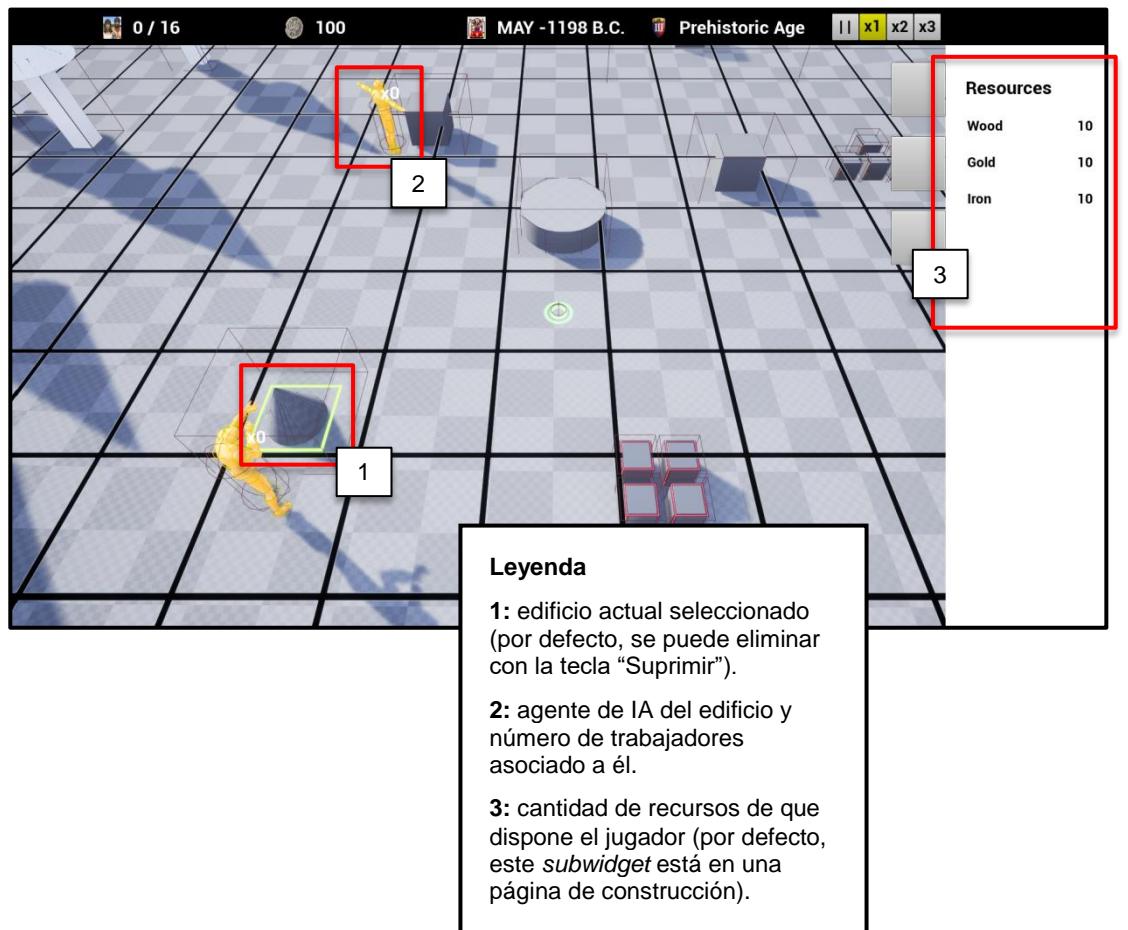
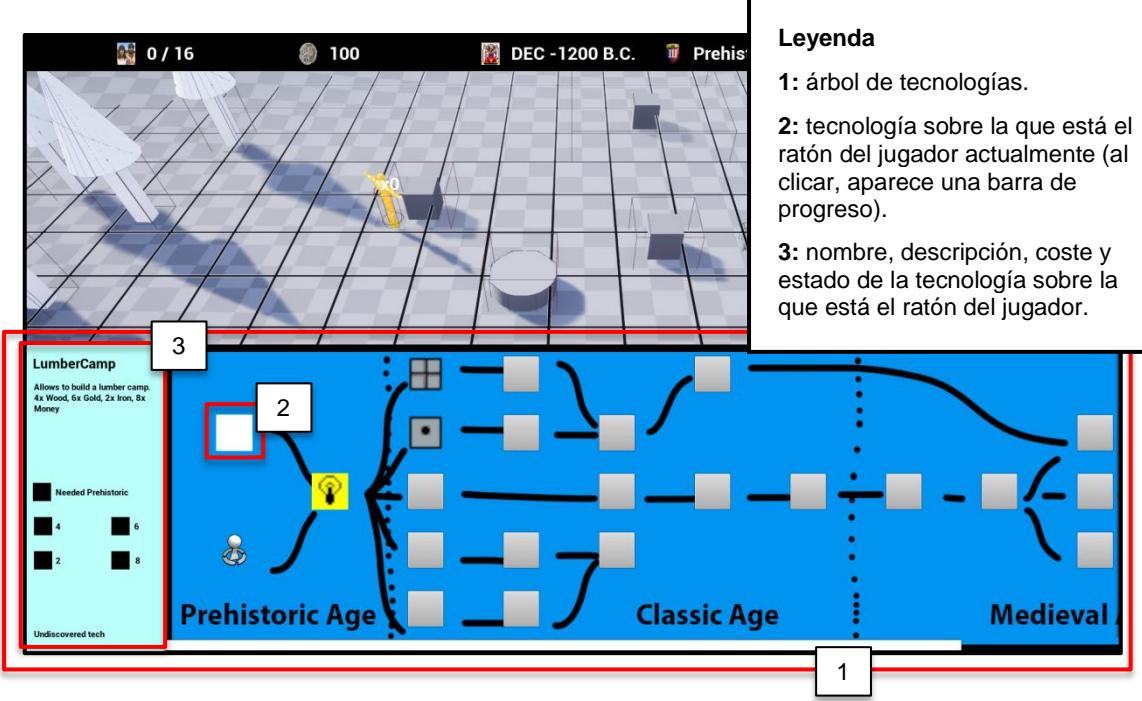


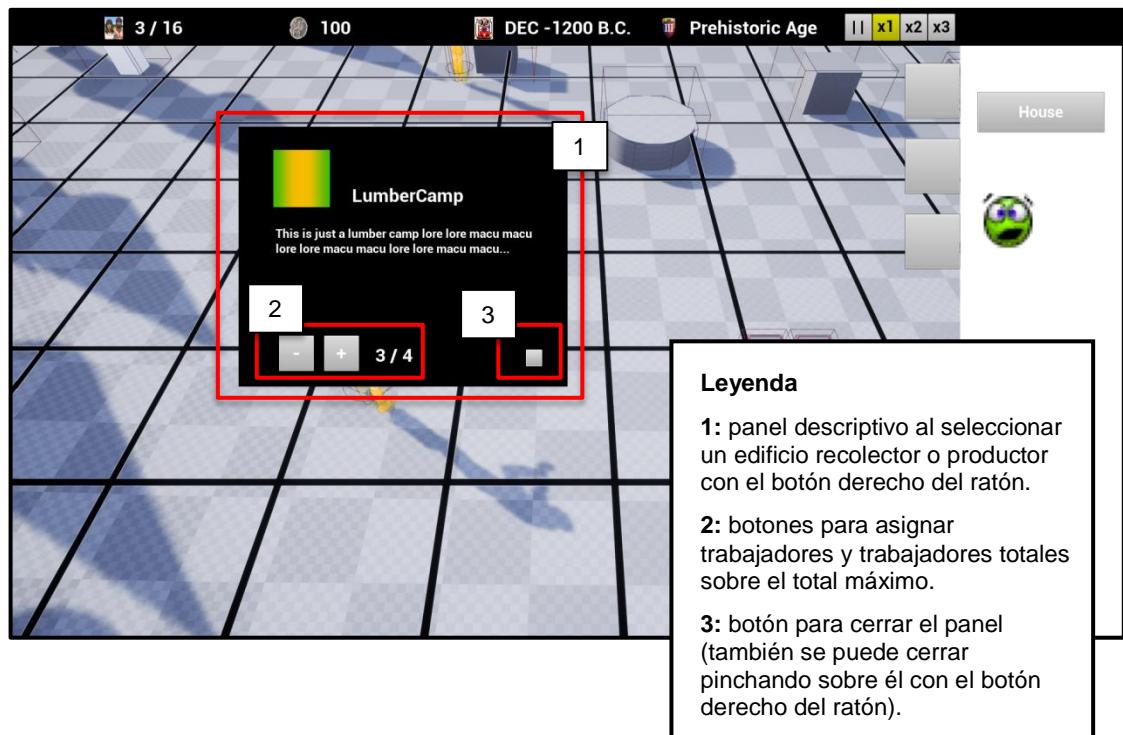
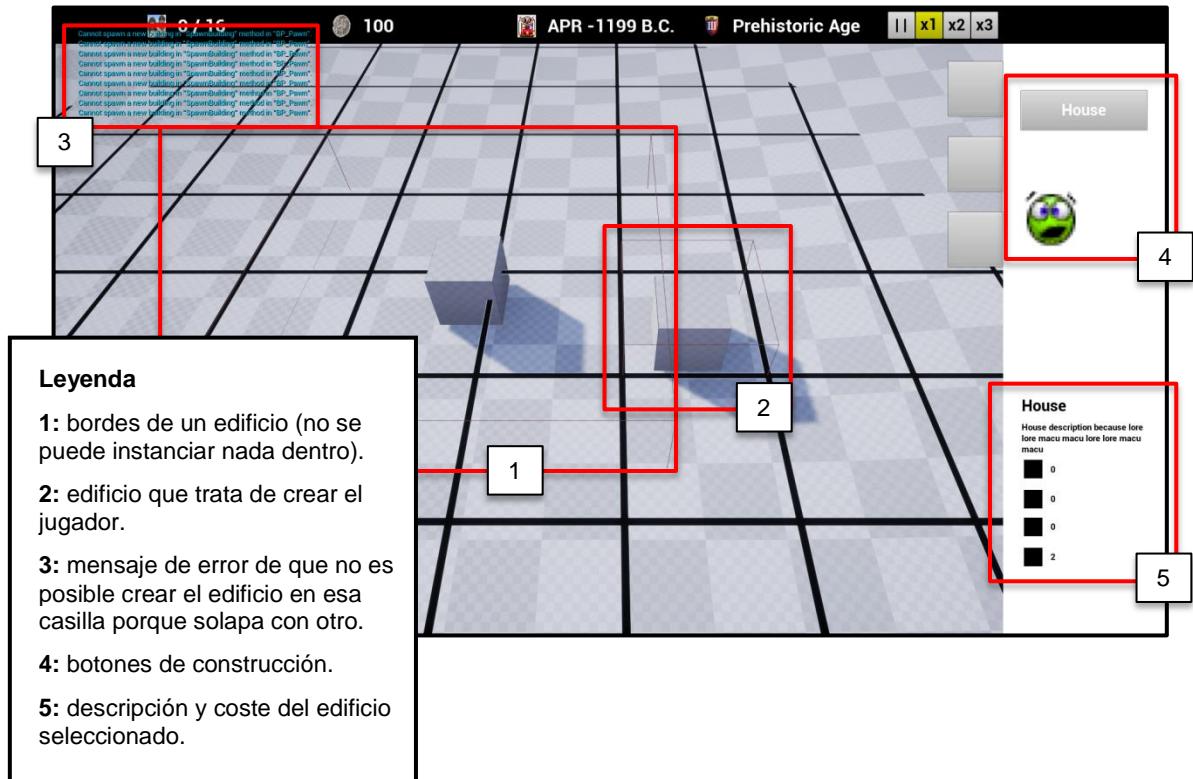
2. Panorámica visual

Las imágenes incluidas en esta sección ofrecen una panorámica visual general del contenido de la plantilla:











The screenshot shows a game interface with a top bar displaying "0 / 16", "100", "SEP -1200 B.C.", "Prehistoric Age", and zoom controls "x1", "x2", "x3". A central black rectangular panel is highlighted with a red border and contains a target icon and the word "House". Below it, text reads: "This is where citizens like Lore and Macu live" followed by the repeating phrase "lore lore macu macu fore fore macu macu". Numbered callouts point to two elements: "1" points to the top right corner of the panel, and "2" points to a small white square button in the bottom right corner of the panel.

Leyenda

1: panel descriptivo al seleccionar un edificio que no sea ni recolector ni productor con el botón derecho del ratón.

2: botón para cerrar el panel (también se puede cerrar pinchando sobre él con el botón derecho del ratón).

The screenshot shows a game interface with a top bar displaying "6 / 16", "104", "FEB -1198 B.C.", "Prehistoric Age", and zoom controls "x1", "x2", "x3". The main view shows several white, leaf-like structures on a grid. Numbered callouts point to four elements: "1" points to a yellow figure carrying a blue resource stack, "2" points to a blue resource stack labeled "x4", "3" points to a yellow figure carrying a smaller blue resource stack, and "4" points to a building labeled "x2".

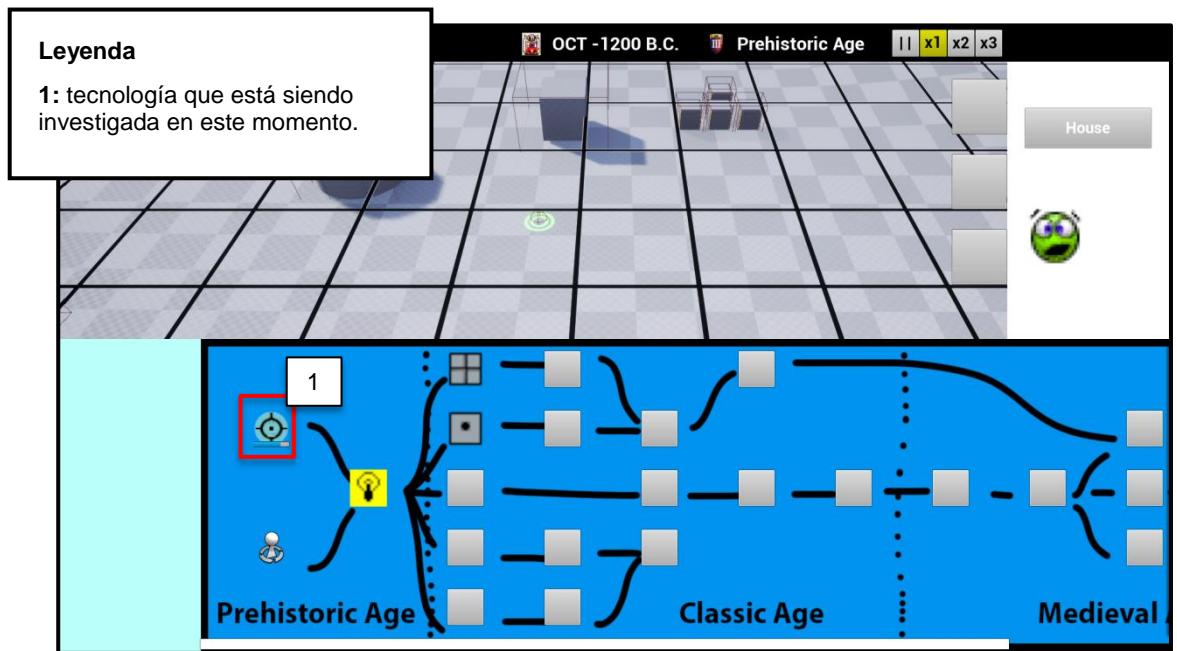
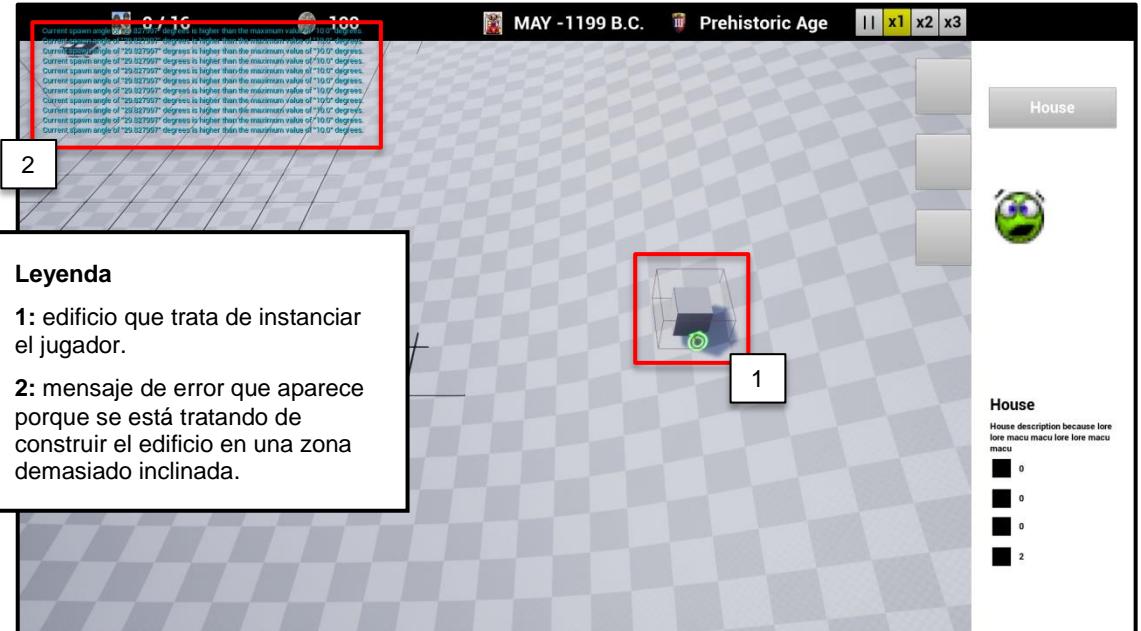
Leyenda

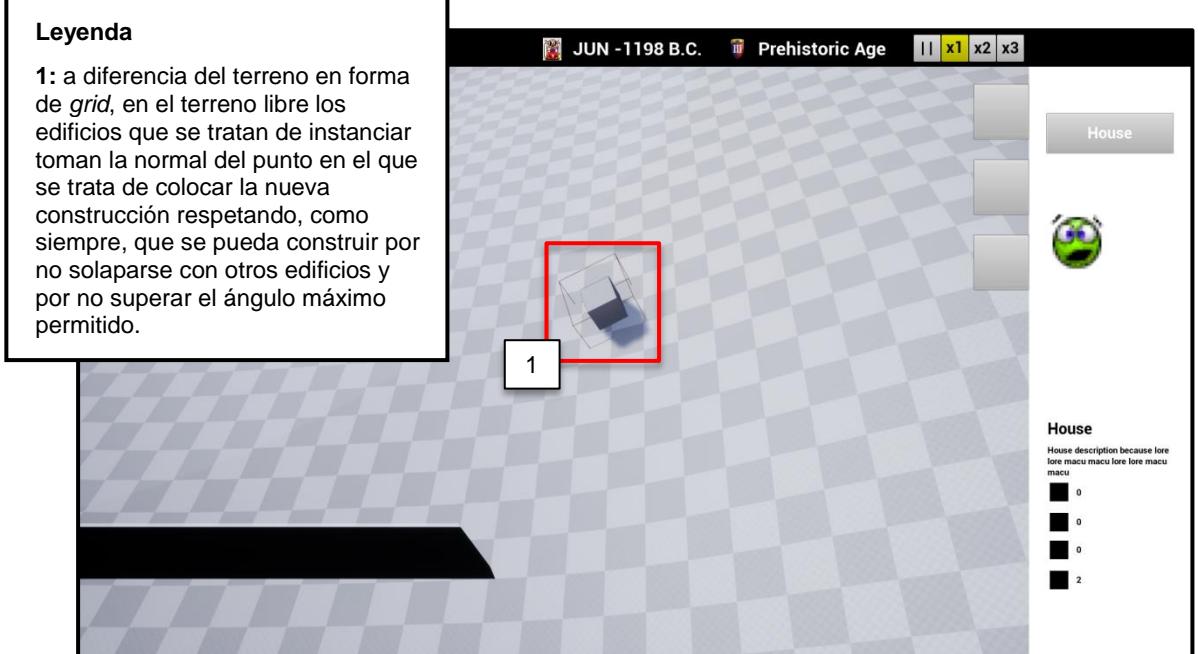
1: recolector que representa a cuatro trabajadores dirigiéndose a la materia prima más cercana.

2: cantidad de recursos que le quedan a la materia prima.

3: productor que representa a dos trabajadores dirigiéndose al almacén más cercano.

4: edificio productor trabajando la materia prima que le entregan los trabajadores.









3. Clases y herencias

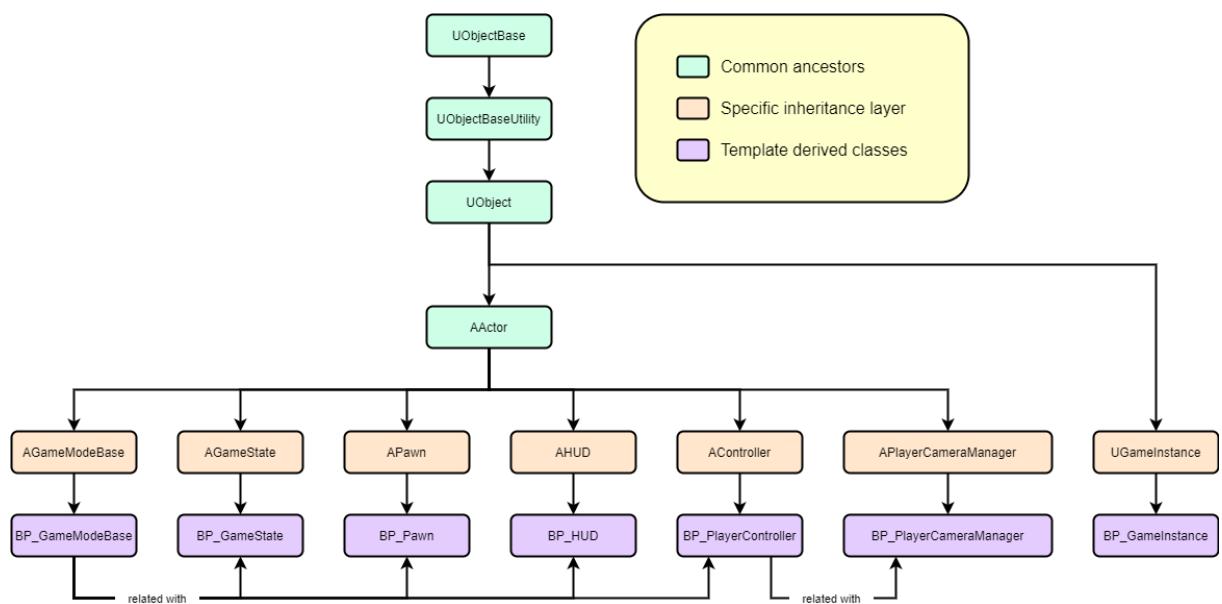
En el presente punto se va a abordar la problemática de las clases implementadas en la plantilla básica, la justificación de su creación y cuál es su jerarquía de herencia. Además, se va a tratar también qué estructuras de datos se han empleado y con qué finalidad.

3.1. Jerarquía de herencias

Este primer subapartado va a revisar la jerarquía de herencias utilizada en las clases ya programadas de antemano junto con la plantilla. Es un rasgo de la programación orientada a objetos muy empleado en la plantilla porque hay clases base en las que hay atributos, interfaces, funciones o métodos que heredan o sobreescriben otras clases derivadas.

3.1.1. Framework principal

El grafo que se muestra en primer lugar ilustra la principal jerarquía de herencia de clases del *framework* principal de la plantilla:



Como se puede observar, la plantilla incluye una serie de clases que conforman el *framework* principal del modo de juego que viene por defecto, reiterar sobre ellas lo siguiente:

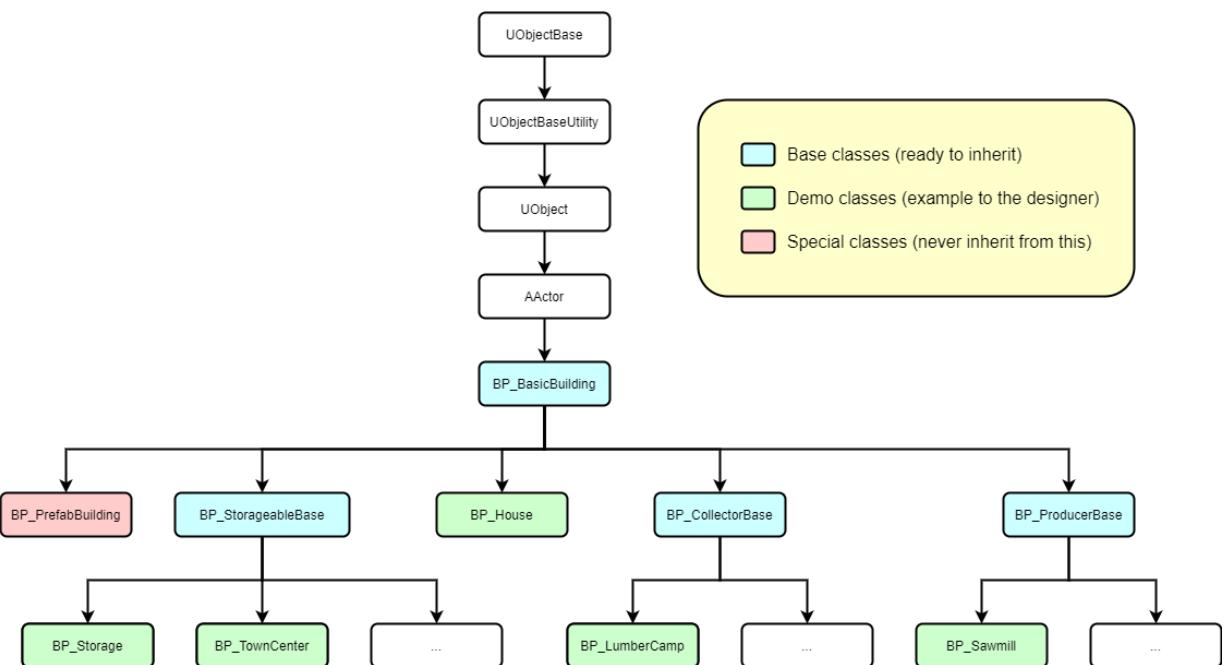
- “BP_GameModeBase”: modo de juego por defecto, relacionado con las clases que se muestran en el diagrama. No hay condiciones de victoria y / o derrota programadas por defecto, es tarea del diseñador configurar los modos de juego necesarios y sus respectivas misiones.
- “BP_GameState”: estado general del juego relacionado con las tecnologías descubiertas, cantidad de recursos que se poseen, población, trabajadores, edad, tiempo del juego, edificios únicos construidos...
- “BP_PlayerController”: canaliza todo el input a la clase “BP_Pawn” excepto el input de pausa, ya especialmente preparado para traspasarlo a la clase “BP_HUD”.



- “BP_Pawn”: personaje jugable implícito, es el encargado de gestionar movimiento del jugador por el mundo e instantiación de enemigos.
- “BP_HUD”: crea el *widget* principal “WB_HUD”. Cualquier menú de pausa debería de crearse aquí.
- “BP_PlayerCameraManager”: ya creado por si el diseñador tuviera necesidad de realizar algo en concreto en la cámara.
- “BP_GameInstance”: por defecto sólo contiene eventos ya que no ha habido necesidad en la plantilla base de darle datos que tengan que persistir de un nivel a otro.

3.1.2. Edificios

El grafo que se muestra en primer lugar ilustra la principal jerarquía de herencia de clases del *framework* principal de la plantilla:



Como se puede observar:

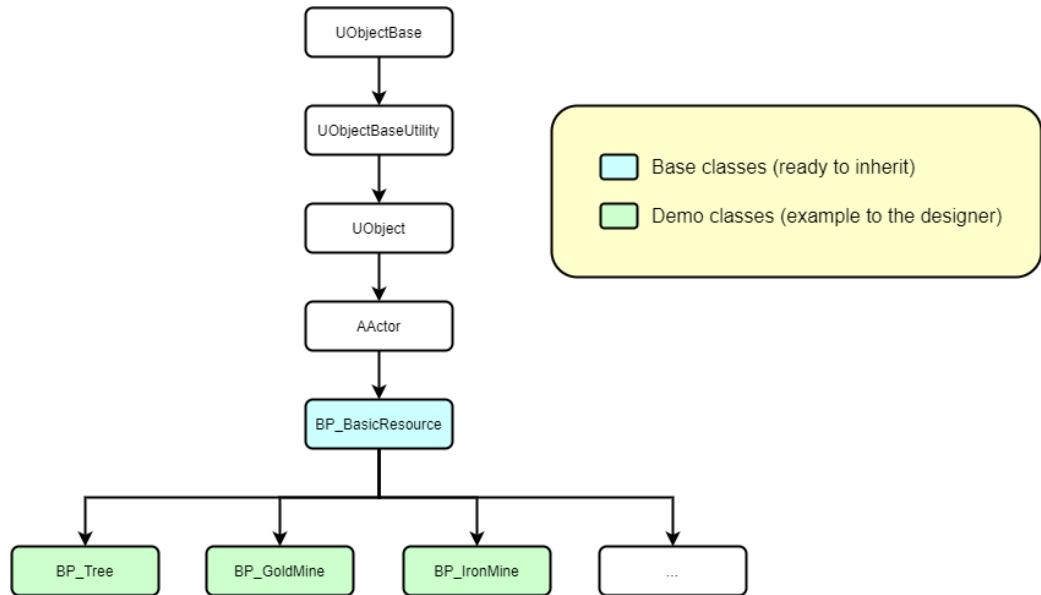
- hay una serie de clases base de las que el diseñador ha de heredar para crear más tipos de edificios;
- existe otro conjunto de clases ya creadas a modo de demo de lo que el diseñador puede hacer;
- y hay incluida otra clase que es especial, de la que no hay que heredar nunca y de la que se hablará en otros epígrafes qué tratamiento se le debe dar.

Se recomienda al diseñador revisar el apartado específico relativo a crear nuevos tipos de edificios, así como revisar la funcionalidad desarrollada en la clase base y en las derivadas que sirven de ejemplo.



3.1.3. Recursos

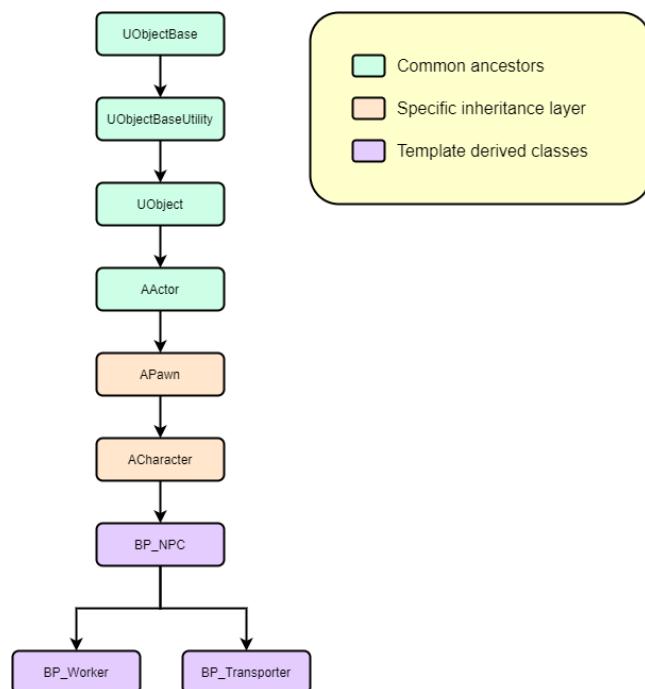
En el siguiente grafo se va a abordar la relación de las clases de recursos ya creadas por defecto junto con la plantilla:



La plantilla incluye una clase actor base y tres clases derivadas a modo de ejemplo. Se recomienda al diseñador revisar el apartado específico relativo a crear nuevos tipos de recursos, así como revisar la funcionalidad desarrollada en la clase base y en las derivadas que sirven de ejemplo.

3.1.4. NPC

Jerarquía de herencia de los agentes de inteligencia artificial:



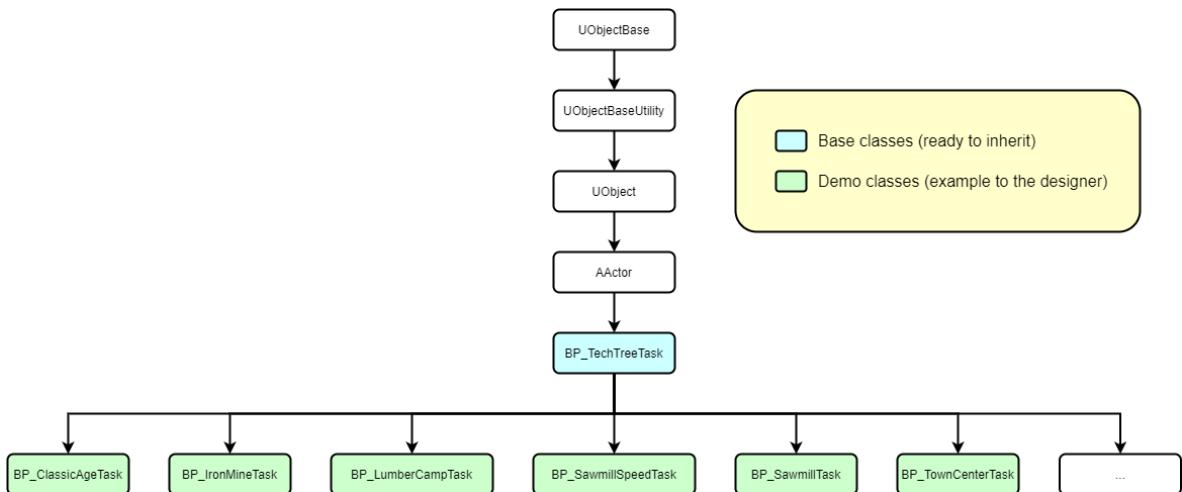


Existe una clase base de nombre “BP_NPC” con atributos y funcionalidad común de la que heredan las otras dos. Por ejemplo, el atributo “resource_multiplier_” en base al número de aldeanos asignados a su respectivo edificio se encuentra aquí, así como el *widget* utilizado como componente para visualizar el número de aldeanos que representa un agente de inteligencia artificial.

Se recomienda al diseñador revisar la funcionalidad desarrollada en la clase base y en ambas derivadas, así como consultar aquellos apartados de la documentación relativos a estas clases.

3.1.5. Tareas del árbol de tecnologías

El siguiente grafo ilustra la jerarquía de clases utilizada en las clases demo del árbol de tecnologías:



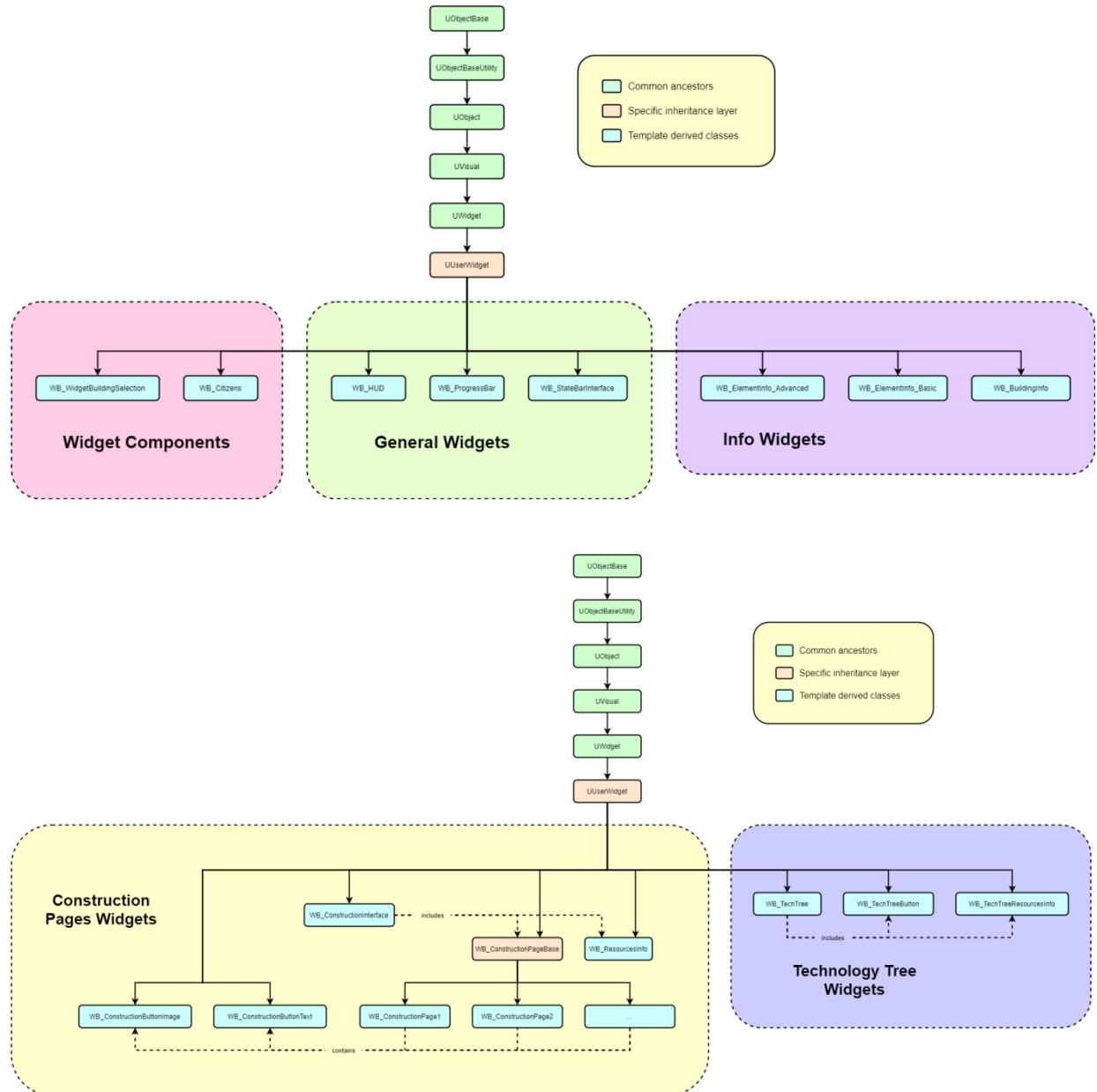
Existe una clase base de nombre “BP_TechTreeTask” con un método de nombre “ExecuteTechTreeTask” que hay que sobrescribir en clases derivadas.

Se recomienda al diseñador revisar específicamente aquellos apartados de la documentación relativos a trabajar con el árbol de tecnologías y a crear nueva funcionalidad en él, así como revisar la funcionalidad desarrollada en la clase base y en las derivadas que sirven de ejemplo.

3.1.6. Widgets

En un juego de estas características existe un abundante uso de *widgets* tanto para la interfaz de usuario como para ser empleados como meros componentes: desde los *widgets* que son utilizados como meros botones de construcción, pasando a los que son empleados para dar información sobre costes de construcción hasta llegar a, por supuesto, aquellos utilizados para representar algo tan complejo de diseñar como pueda ser un árbol de tecnologías.

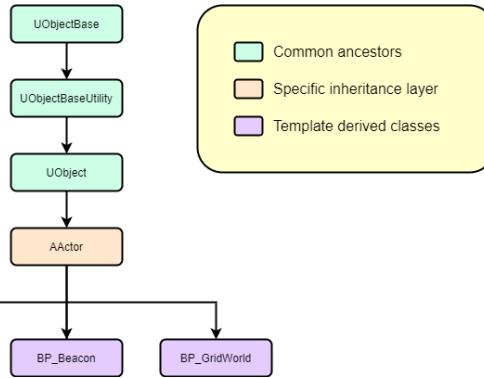
Los siguientes grafos ilustran la jerarquía de *widgets* utilizada en la plantilla, se han tenido que emplear un par de imágenes debido a la gran cantidad de clases a documentar:



Se recomienda al diseñador revisar la funcionalidad desarrollada en cada una de estas clases, así como consultar aquellos apartados de la documentación relativos a los distintos sistemas en los que son utilizados.

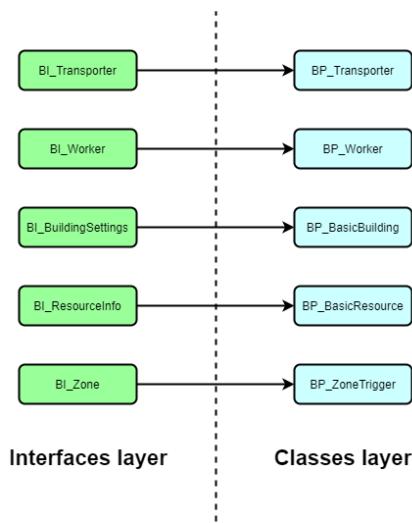
3.1.7. Otras clases

Para finalizar este primer apartado, en la plantilla existen otras clases que no guardan relación con las jerarquías anteriormente mostradas pero que son de vital importancia tanto para el diseñador como para el jugador. Son las que se muestran en el siguiente grafo de actores:



3.2. Interfaces

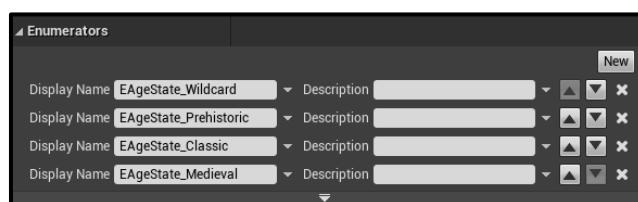
Existen una serie de interfaces en la plantilla que se emplean dentro de los siguientes arquetipos:



Se recomienda al diseñador echar un vistazo a las funciones que incluye cada interfaz, así como al resto de la documentación porque algunas de ellas ofrecen puertas de entrada a métodos que el diseñador ha de reimplementar.

3.3. Enumeraciones

Existen una serie de “enums” creadas por defecto en el proyecto que el desarrollador ha de modificar a su gusto. La plantilla incluye, en primer lugar, la “enum” de nombre “EAgeState” con una serie de edades de base que se pueden modificar, añadir o eliminar:





Lo mismo sucede con la “enum” de nombre “EBuildingType” ya que esta enumeración tiene que recoger todos los edificios planteados por el diseñador ya que luego esos valores son empleados en los distintos sistemas de la plantilla. Hay una serie de edificios por defecto que el diseñador puede modificar según sus necesidades:

Enumerators	
Display Name	EBuildingType_Undefined
Description	
Display Name	EBuildingType_TownCenter
Description	
Display Name	EBuildingType_House
Description	
Display Name	EBuildingType_Storage
Description	
Display Name	EBuildingType_LumberCamp
Description	
Display Name	EBuildingType_Sawmill
Description	
Display Name	EBuildingType_GoldMine
Description	
Display Name	EBuildingType_IronMine
Description	

Del mismo modo, el enum” de nombre “EResourceType” ha de recoger todos los tipos de recursos disponibles en la plantilla, sean materias primas o recursos manufacturados. De nuevo hay una serie de valores por defecto que pueden ser modificados:

Enumerators	
Display Name	EResourceType_Wood
Description	
Display Name	EResourceType_Iron
Description	
Display Name	EResourceType_Gold
Description	

En cambio, la “enum” de nombre “EWorldType” se trata de una “enum” más de uso interno para ejecutar una funcionalidad u otra según el diseñador le diga a su mundo de juego, personificado en la instancia de la clase “BP_GridWorld”, que este es libre o en grid:

Enumerators	
Display Name	EWorldType_Free
Description	
Display Name	EWorldType_SquaredGrid
Description	

Finalmente, la “enum” “EZoneType” sirve para asociarla triggers que delimiten la zona incluida en ellos como una zona de un tipo en concreto en la que hayan una serie de edificios que sólo se puedan instanciar en ella –aquellos que comparten tipo o sean del tipo “wildcard”–, marcado así individualmente en cada uno de ellos:

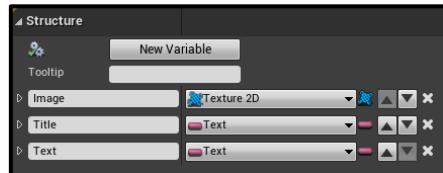
Enumerators	
Display Name	EZoneType_Wildcard
Description	
Display Name	EZoneType_River
Description	
Display Name	EZoneType_Farm
Description	
Display Name	EZoneType_Forest
Description	

3.4. Estructuras de datos

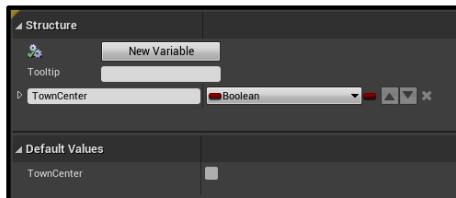
Por lo que se refiere a las estructuras de datos, a lo largo de la plantilla hay algunos tipos definidos con importancia core. A destacar, en primer lugar, la estructura de datos de nombre “**S_DescriptionInfo**”, estructura de la que se han declarado atributos en las



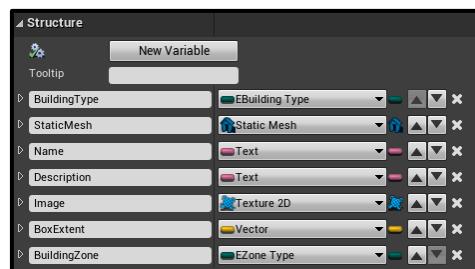
clases “BP_BasicResource” y “BP_BasicBuilding” para mostrar información sobre cada recurso o edificio en el momento en el que el usuario pinche sobre él con el botón derecho del ratón. En principio no se atisba necesario que el diseñador tenga que añadir ningún dato más en esta estructura:



En segundo lugar tenemos la estructura de datos “**S_UniqueBuilding**” donde el usuario creará *flags* para todos los edificios únicos que vaya a tener el juego porque, como se especificará en su respectivo apartado, de esta estructura el juego leerá si un edificio único ha sido o no construido:



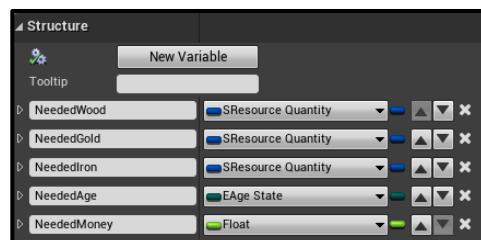
La estructura de datos “**S_BuildingInfo**” se emplea, en cambio, para que cada botón de construcción sepa qué tipo de edificio se ha seleccionado para comunicar información al jugador y previsualizar el nuevo edificio en vivo mediante la instancia del nivel de la clase “BP_PrefabBuilding”:



Hay más estructuras en la plantilla como la estructura “**S_ResourceQuantity**”...



... que se emplea como dato en “**S_ElementCost**” para definir el coste de cada tipo de edificio y tecnología:





Otra estructura es la estructura “**S_ResourceInfo**” que se emplea en los actores que heredan de “BP_BasicResource” para indicarle cómo funciona:

Structure	
	New Variable
Tooltip	
↳ Type	EResource Type
↳ Quantity	Float
↳ QuantityPerTask	Float
↳ InitialQuantity	Float
↳ TaskCooldown	Float
↳ TaskMultiplier	Float
↳ RespawnCooldown	Float
↳ ShouldRespawn	Boolean
↳ IsAlive	Boolean

Finalmente, la plantilla incluye la estructura “**S_TechInfo**” para determinar todas las características de las tecnologías del árbol de tecnologías:

Structure	
	New Variable
Tooltip	
↳ Name	Text
↳ Description	Text
↳ Image	Texture 2D
↳ CompletedImage	Texture 2D
↳ TimeToComplete	Float
↳ Cost	SElement Cost
↳ TaskClass	BP Tech Tree Task

... y la estructura “**S_TechCollection**” para almacenar internamente si hay edificios nuevos descubiertos por alguna tecnología o si se ha investigado alguna otra que permite acelerar algún proceso de producción en el edificio que sea:

Structure	
	New Variable
Tooltip	
↳ TownCenter	Boolean
↳ House	Boolean
↳ Storage	Boolean
↳ LumberCamp	Boolean
↳ Sawmill	Boolean
↳ GoldMachinery	Boolean
↳ IronMachinery	Boolean
↳ SawmillSpeed	Float
↳ SawmillGold	Float
↳ WorkerDelayInBuilding	Float

3.4.1. Taxonomía

De las estructuras de datos arriba enunciadas, hay unas que son más de uso interno y, por lo tanto, potencialmente intocables para el usuario: “S_DescriptionInfo”, “S_BuildingInfo”, “S_ResourceQuantity”, “S_ResourceInfo” y “S_TechInfo”, estructuras que se utilizarán donde sea con la información que sea.

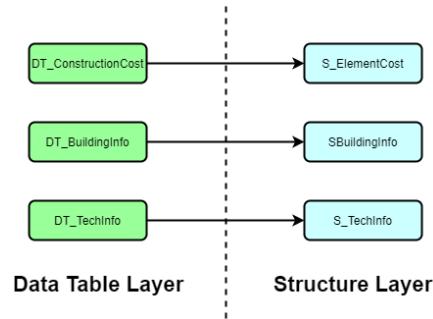
No obstante, existe una serie de estructuras que sí han de ser modificadas por el diseñador y que, en el momento de la entrega de la plantilla, contienen datos netamente de ejemplo: “S_UniqueBuilding”, “S_ElementCost” y “S_TechCollection”.



Se recomienda al desarrollador consultar el apartado de “Modificación de la plantilla” para seguir los tutoriales concretos relacionados con cada apartado del juego pero, sobre todo, consultar el subapartado de nombre “Modificación de las estructuras de datos”.

3.5. Tablas de datos

Para finalizar este epígrafe, es esencial resaltar que destacar que hay un total de tres *data table* relacionadas cada una con su respectiva estructura de datos de la siguiente manera:



De estas tres, las *data table* de nombre “DT_ConstructionCost” y “DT_BuildingInfo” han de tener los mismos “RowName” porque están relacionadas entre sí.



4. Flow del input

En esta sección se va a detallar cómo es el proceso de input desarrollado para la presente plantilla: la configuración predeterminada enlaza los “Action Mappings” y los “Axis Mappings” incluidos en “Project Settings” con eventos de input dentro de la clase “BP_PlayerController”. Desde esta clase se enlaza, en la mayoría de ocasiones, con métodos de la clase “BP_Pawn” excepto por el evento “PauseAction”, especialmente preparado en el “Player Controller” para que el diseñador cree funcionalidad relacionada con un menú de pausa.

Los “Action Mappings” y “Axis Mappings” que vienen por defecto con la plantilla son los siguientes:







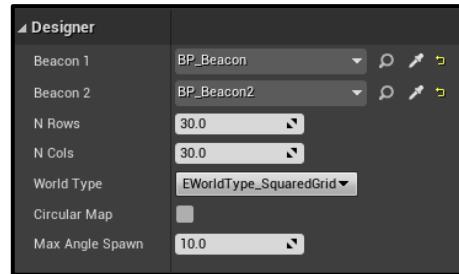
5. Modificación de la plantilla

El objetivo de esta sección es clarificar al diseñador puntos importantes sobre el contenido entregado en la plantilla para que pueda realizar modificaciones rápidas según sus necesidades proyectuales.

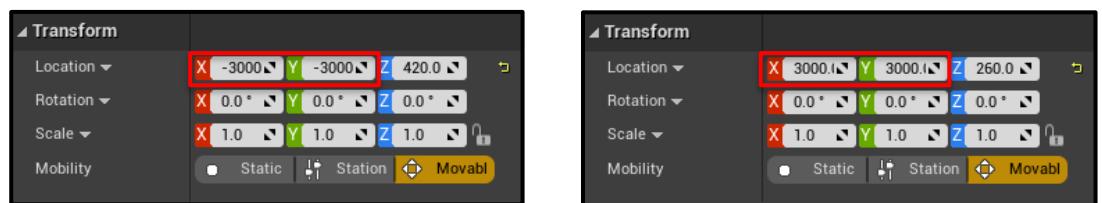
5.1. Configuración en edición de “BP_GridWorld”

A la hora de crear un nivel de juego, el diseñador ha de tener en cuenta las siguientes consideraciones en tiempo de edición:

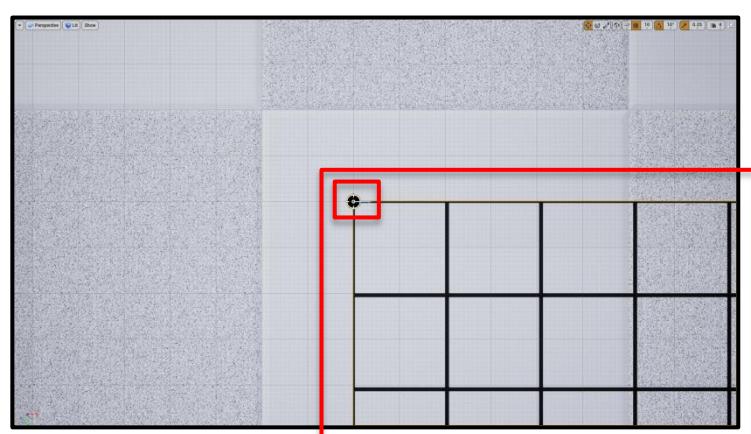
1. Determinar si el terreno de juego va a ser sobre un **plano o** sobre una **landscape**.
2. Instanciar un **actor de “BP_GridWorld”** centrado en torno al “0.0f, 0.0f” en los ejes de “X” e “Y” sobre el mundo de juego, con el escalado que sea en los mismos ejes –aunque esto es algo más bien visual por ahora– y ajustar sus atributos:



3. Nótese que a este actor vamos a asignarle un **par de actores de la clase “BP_Beacon”** que instanciaremos manualmente en las esquinas de nuestro mundo, uno para que tome valores mínimos en los ejes de la “X” e “Y” y otro para que tome valores máximos en los mismos ejes.

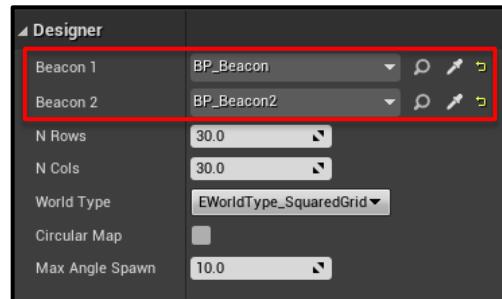


4. Recalcar que estos actores baliza **los colocaremos manualmente en sendas esquinas de mínimos y máximos** de nuestro actor de la clase “BP_GridWorld”:

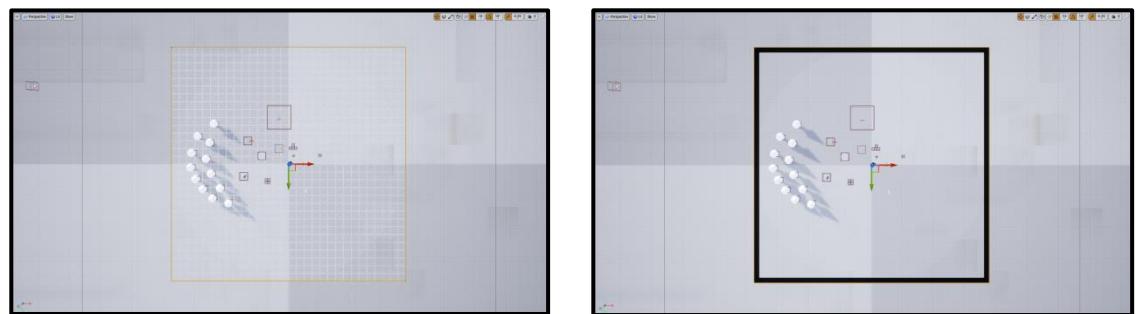




5. De los atributos anteriormente señalados, en primer lugar **asignamos los actores baliza** que acabamos de instanciar:

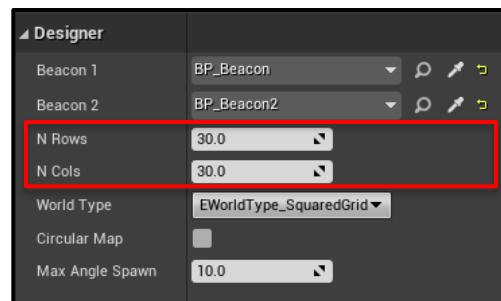


6. Ahora es el turno de decidir si nuestro mundo va a ser un **mundo de juego en forma cuadrangular o de instantiación libre**. La clase "BP_GridWorld" tiene un plano que es insensible a todo raycast o colisión porque la única utilidad de este "UStaticMeshComponent" es visual ya que emplea un material para señalar visualmente al jugador con qué tipo de mundo estamos trabajando:



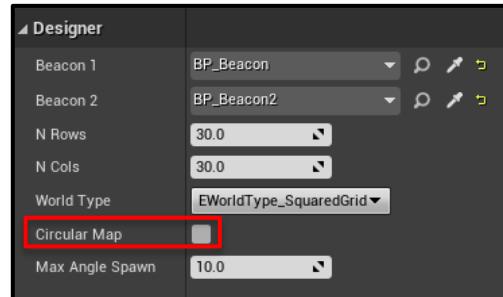
7. En el caso de haber determinado que la instantiación de los edificios sea sobre una *grid cuadrangular*, hemos de determinar a continuación el número de filas y de columnas de esta parrilla. En el supuesto de que quisiéramos de que los edificios normales, la gran mayoría, ocupen lo que ocupa una posición del mundo, sólo en ese supuesto se recomienda que estos dos valores se correspondan *grosso modo* con los *bounds* o límites de nuestros edificios en los ejes "X" e "Y" –es posible que un edificio ocupe varias posiciones, pero eso se tratará en otro apartado–.

Para nuestro ejemplo, en el que teníamos las balizas en “-3000.0f” y “3000.0f” respectivamente en ambos ejes, para unas dimensiones de juego totales de “6000.0f” unidades en ambos ejes, si determinamos que el número de filas sea de “30.0f” y el número de columnas también sea de “30.0f”, cada casilla ocupa “200.0f” unidades de “UE4” en ambos ejes –imprescindible recordar que una unidad de “UE4” equivale a un centímetro de la vida real–:

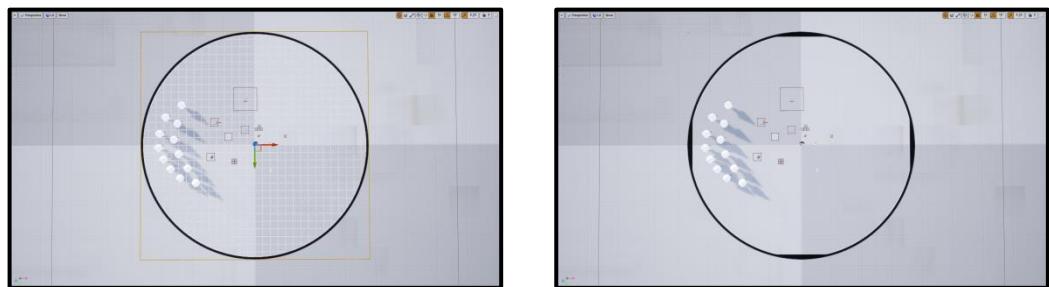




8. Por el contrario, estos dos atributos darán absolutamente igual en el supuesto de que el mapa sea circular:

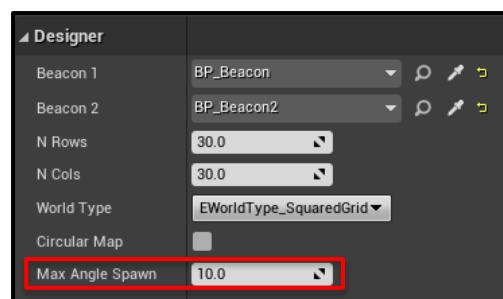


9. En ese supuesto, es decir, cuando el mapa sea circular es cuando el mundo de juego se renderizará como se muestra en las siguientes imágenes. A destacar que los parámetros que necesita este material se pasan en el "Construction Script" de la clase "BP_GridWorld":



10. Es reseñable destacar, tal y como se ha comentado en epígrafes anteriores, que en el supuesto de que el **mundo de juego** sea **circular** –independientemente de que sea libre o en **grid**– sólo se tomará en cuenta la **posición de la primera baliza** sobre el mundo de juego y **en términos absolutos** –sin valores negativos–.

11. Finalmente, tener en cuenta el atributo “Max Angle Spawn” para determinar el ángulo máximo al que nuestro personaje jugable podrá instanciar un nuevo edificio en el mundo:

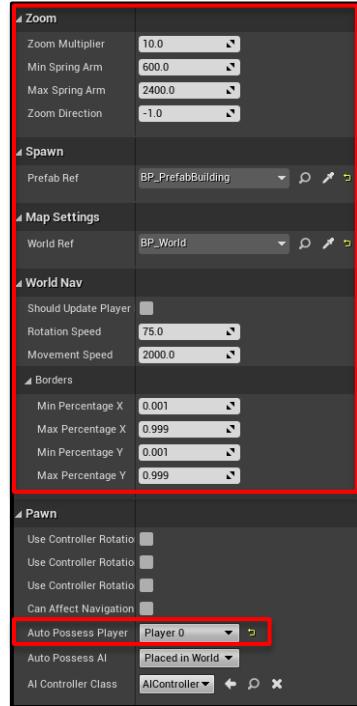


5.2. Terminar de preparar el mundo de juego

Una vez configurado el actor de la clase “BP_GridWorld”, que aunque parezca una malla es más funcional que estético, es hora de terminar de preparar el mundo de juego:



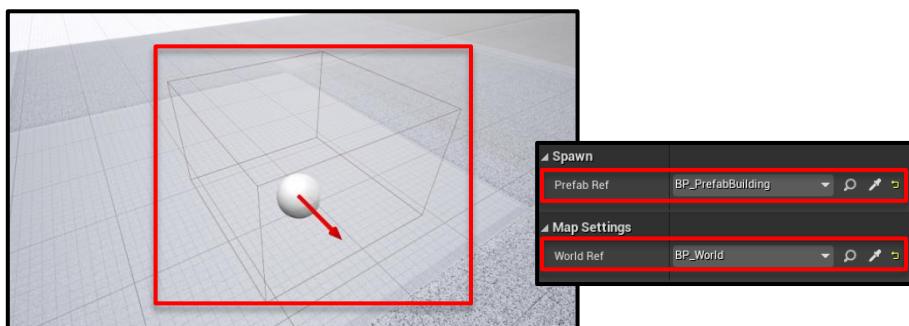
1. Instanciar un **actor de la clase “BP_Pawn”** por el centro del mundo de juego y ajustar los siguientes valores:



2. Como se puede observar, hay **varios epígrafes** en los que cada uno sirve para una cosa: las variables del epígrafe “Zoom” para modificar ajustes relativos al zoom como la dirección, el multiplicador para ajustar velocidad y un mínimo y un máximo de altura absoluta –no relativa–; una referencia a un actor de la clase “BP_GridWorld” y a otro de la clase “BP_PrefabBuilding” previamente instanciados en el nivel –en este punto el primero ya estará instanciado pero no todavía el segundo–; velocidad de movimiento y de rotación, y cuándo detectar un borde en términos porcentuales de la posición del ratón sobre la ventana gráfica.

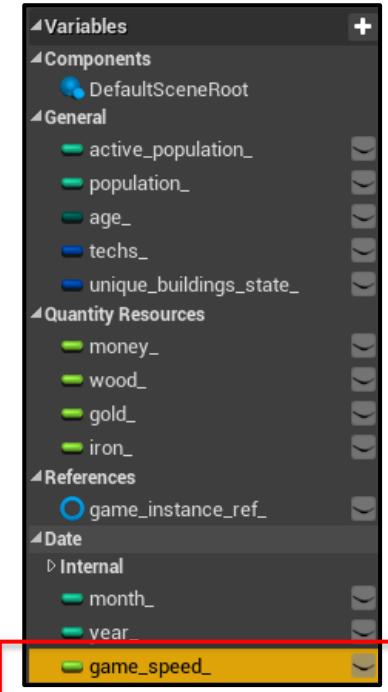
La **altura a la que coloquemos el “pawn”** en el eje de la “Z” **repercute**, obviamente, **sobre lo cercano o lejano del zoom**: los otros valores son relativos a esta altura porque agrandan o achican la longitud del “SpringArmComponent” de la clase del jugador.

3. Por ello, a continuación se instanciará un **objeto de la clase “BP_PrefabBuilding”** que encontramos dentro de “/Content/StudentName/Placeables/Buildings” y se lo asignaremos a nuestra instancia de “BP_Pawn” en el mundo porque este actor actuará como plantilla para que el jugador pueda previsualizar cada nuevo edificio que vayamos a instanciar:



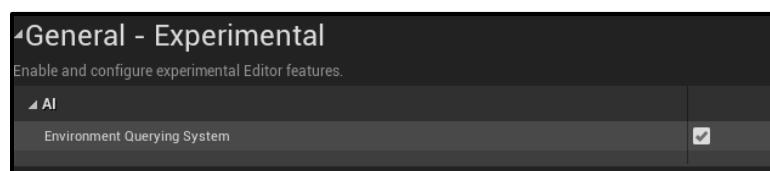


4. Colocar manualmente sobre el propio nivel los **recursos** con los que se quiera jugar en el nivel en concreto, así como los **edificios** que se quieran otorgar de entrada al jugador.
5. Finalmente, en la clase “BP_GameState”, terminar de **configurar la velocidad de juego** en la variable “game_speed_”, esto es: ¿cuántos segundos de la vida real ha de durar un mes del propio juego?



5.3. Revisar el estado del EQS

Como los agentes de inteligencia artificial de la plantilla hacen uso de la tecnología experimental EQS (*environment query system*), y pese a que en el momento de desarrollar la plantilla se encuentra activado en el motor, es importante que el diseñador revise que la **flag “Environment Querying System”** que encontramos en “Editor Preferences” se encuentre activada como se muestra en la siguiente imagen:



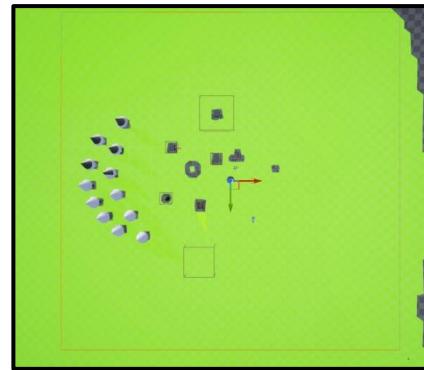
5.4. Tener lista la “Navigation Mesh”

Simplemente a modo de recordatorio, es importante recalcar que los agentes de inteligencia artificial no van a funcionar sin un actor “Navigation Mesh” instanciado en el mundo.

Es recomendable que tenga la **dimensión suficiente** para que cubra toda el área que abarca el actor de la clase “BP_GridWorld” y los dos “BP_Beacon” instanciados en sus



esquinas, la tecla para observar la zona navegable en tiempo de edición sigue siendo la tecla 'P' por defecto:



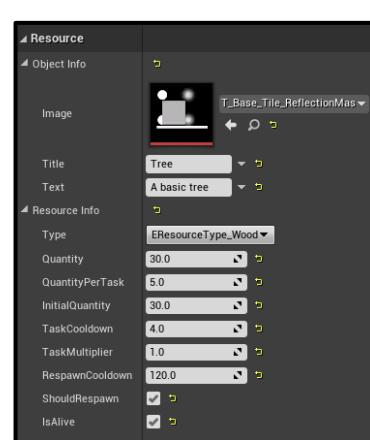
5.5. Añadir recursos de materias primas

La plantilla entregada cuenta con tres recursos esenciales que son: madera, hierro y oro. Adicionalmente existe el recurso monetario, aunque no es exactamente una materia prima por lo que no es objetivo exclusivo de este epígrafe. A la hora de añadir nuevos recursos de materias primas o modificar los ya existentes, el diseñador ha de hacer lo siguiente:

1. Crear el **tipo de la nueva materia prima** en el “enum” “**EResourcetype**” que encontramos en la ruta “/Content/StudentName/Placeables/Resources”.



2. Crear una **nueva clase** que herede de la clase actor “**BP_BasicResource**”, clase localizada en la misma carpeta que el “enum” anterior.
3. En la clase derivada recién creada, el diseñador tiene que **modificar la información relativa a los atributos “object_info_” y “resource_info_”** que hereda de la clase padre. El primer atributo es información para mostrar al jugador cuando pinche con el botón derecho del ratón sobre el recurso y el segundo atributo permite determinar el tipo de recurso, unidades del recurso y unidades que da a los trabajadores cuando van a él, si ha de regenerarse y tiempo que tarda en hacerlo...

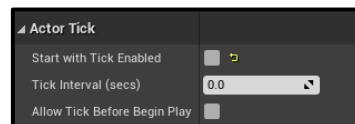




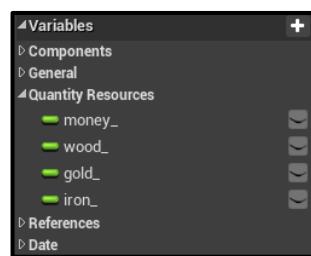
4. La razón de que sean **atributos editables desde la pestaña principal** del motor es que podemos dar una información general a nivel de clase pero luego modificar en concreto los atributos de una instancia para hacer que, por ejemplo, un árbol comience dando la mitad de recursos que puede dar.
5. Una vez realizados estos cambios en la nueva clase hija creada, únicamente quedaría por **asignar geometría a la “StaticMeshComponent”** que la forma, **ajustar tamaños de la caja de colisión y recolocar** algún que otro componente como **la “QuantityProgressBar”**.



6. Destacar que, a priori, no habría que escribir funcionalidad en clases derivadas de “BP_BasicResource” puesto que **toda la funcionalidad ya está creada en la clase base** –incluyendo el uso de una interfaz de nombre “BI_Resourcelnfo”– y, por herencia, utilizada en clases hijas. Comentar también que el “Tick” está desactivado en la clase padre, y este rasgo lo heredan las clases hijas.



7. Ahora hay que acudir a la clase “BP_GameState” que encontramos en la ruta “/Content/StudentName/Framework” para, por una parte, **añadir una variable de tipo float en el epígrafe “QuantityResources”** para almacenar la cantidad de recursos de que dispondremos en el juego del nuevo recurso...:

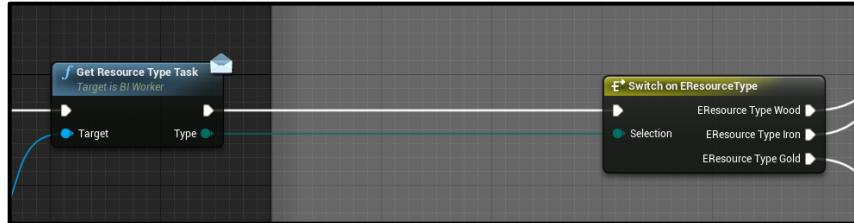


...y **adaptar una serie de métodos de “BP_GameState”**: “UpdateResource”, “CheckIfResourcesAndAgeAreAvailable” y “UpdateResourcesWhenPurchasing”.

8. Además, el diseñador ha de recordar también **modificar los siguientes widgets**:
 - “WB_BuildingInfo” en “/Content/StudentName/UI/Info”.
 - “WB_ResourcesInfo” en “/Content/StudentName/UI/ConstructionPages”.
 - “WB_TechTreeResourcesInfo” en “/Content/StudentName/UI/TechnologyTree”.
9. Así como **actualizar los tipos de recursos de la estructura “S_ElementCost”** que hay en “/Content/StudentName/Placeables/Buildings” –revisar el apartado de la documentación “Modificación de las estructuras de datos”–.



10. Para finalizar, el diseñador ha de **actualizar el método “DoltemGeneration”** de “EQG_SearchResources” con ruta en “/Content/StudentName/NPC/Worker” porque ahí hay un *switch* en el que rehacer funcionalidad con los nuevos tipos de recursos que hay:



5.6. Añadir recursos manufacturados

En cambio, si deseamos añadir un recurso manufacturado como, por ejemplo, puedan ser tablones de madera en base a la materia prima de la madera, los pasos se limitan a prácticamente los últimos pasos del epígrafe anterior:

1. No hay que crear **ninguna clase de actor nueva**.
2. Hay que acudir de nuevo a la clase “BP_GameState” que encontramos en la ruta “/Content/StudentName/Framework” para, por una parte, añadir una variable de tipo *float* en el epígrafe “QuantityResources” para almacenar la cantidad de recursos de que dispondremos en el juego del nuevo recurso...:



...así como **adaptar** funcionalidad incluida en algunos **métodos de la clase “BP_GameState”**: “UpdateResource”, “CheckIfResourcesAndAgeAreAvailable” y “UpdateResourcesWhenPurchasing”.

3. Además, el diseñador ha de recordar también **modificar los siguientes widgets**:
 - “WB_BuildingInfo” en “/Content/StudentName/UI/Info”.
 - “WB_ResourcesInfo” en “/Content/StudentName/UI/ConstructionPages”.
 - “WB_TechTreeResourcesInfo” en “/Content/StudentName/UI/TechnologyTree”.
4. Así como **actualizar los tipos de recursos de la estructura “S_ElementCost”** que encontramos en “/Content/StudentName/Placeables/Buildings” –revisar el apartado de la documentación “Modificación de las estructuras de datos”–.

5.7. Generar recursos monetarios

El recurso monetario se generará potencialmente en edificios productores, por lo que es **decisión del diseñador** determinar qué edificios van a generar monedas y de qué



forma van a hacerlo: es uno de los puntos que el diseñador ha de escriptar y programar. En el momento de entrega de la plantilla, cuenta con un ejemplo ya desarrollado en la clase “BP_Sawmill”.

5.8. Añadir nuevos tipos de edificios

La plantilla permite añadir tantos tipos de edificios como el diseñador necesite, pero para ello ha de seguir unos pasos en concretos para aprovechar la funcionalidad base de la plantilla:

1. En primer lugar, hay que **añadir el nuevo tipo de edificio en** la “enum” de nombre **“EBuildingType”** que hay en la ruta “/Content/StudentName/Placeables/Buildings”. Los siguientes tipos son los que vienen por defecto por la plantilla, siendo susceptibles de sufrir cualquier cambio por parte del diseñador:

Enumerators	
Display Name	EBuildingType_Undefined
Display Name	EBuildingType_TownCenter
Display Name	EBuildingType_House
Display Name	EBuildingType_Storage
Display Name	EBuildingType_LumberCamp
Display Name	EBuildingType_Sawmill
Display Name	EBuildingType_GoldMine
Display Name	EBuildingType_IronMine

2. A continuación vamos a **crear la nueva clase de edificio**. Para ello, y a modo de avance de lo que se explicará en los siguientes subapartados, heredaremos de una de las siguientes clases según lo que queramos crear:

- **Recolector:** edificio que mandará un trabajador al recurso a por la materia prima, heredaremos directamente de la clase “BP_CollectorBase” –ruta en la carpeta “/Content/StudentName/Placeables/Buildings/Economy”–. La plantilla incluye, a modo de ejemplo, un campamento maderero que recolecta madera de los árboles.
- **Productor:** edificio que mandará un transportista al almacén más cercano a por el recurso con el que vaya a trabajar el edificio industrial en cuestión, heredaremos directamente de la clase “BP_ProducerBase” –ruta en la carpeta “/Content/StudentName/Placeables/Buildings/Economy”–. La plantilla incluye, a modo de ejemplo, una serrería que convierte madera en monedas.
- **Almacén:** edificio del que heredarán todos aquellos edificios a los que acudirán los transportistas, heredaremos directamente de la clase “BP_StorageableBase” con ruta en “/Content/StudentName/Placeables/Buildings/Storageables”. En la demo, el centro urbano hereda de esta clase y además es un edificio único en el momento de la entrega al diseñador. Además de este edificio, la plantilla incluye, a modo de ejemplo, un almacén básico.
- **Cualquier otro tipo de edificio** heredará de la clase “BP_BasicBuilding”, con ruta en la carpeta “/Content/StudentName/Placeables/Buildings”. Las casas programadas en la demo, por ejemplo, heredan directamente de esta clase padre.

3. El hecho de heredar de según qué clase para el tipo de edificio que se desea construir viene motivado por el hecho de que estas clases ya incluyen una **cierta funcionalidad básica**:

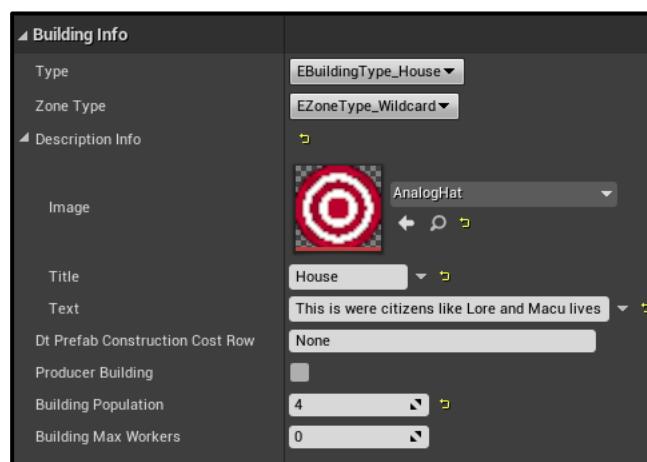
- la **interfaz “BI_BuildingSettings”** que aporta una serie de prototipados de funciones que reimplementar en caso de necesidad;



- una serie de **atributos** a los que dar valor –siguiente punto–;
- y un conjunto de **métodos** que se pueden reutilizar o sobreescribir para reimplementar con funcionalidad específica por parte del desarrollador.

4. Una vez se ha creado la nueva clase, se procede a **modificar los atributos del epígrafe “BuildingInfo”** que se muestran en la siguiente captura:

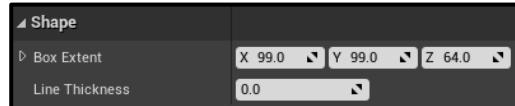
- **tipo de edificio** correspondiente con un valor recogido en el “enum” de nombre “EBuildingType” mentado líneas atrás;
- **tipo de zona** en la que se puede construir –si el edificio se puede construir en cualquier lugar del mundo, su tipo será el tipo “wildcard”–, valor que tiene su correspondencia con el “enum” de nombre “EZoneType”;
- **información** detallada con imagen, nombre y descripción para el momento en el que el jugador pinche con el botón derecho del ratón sobre el edificio;
- el **coste de construcción** sirve para que el actor de la clase “BP_Pawn” conozca, en su método “SpawnBuilding”, cuánto va a costar producir este edificio en el momento en el que el usuario trate de instanciarlo en el mundo en base a información de la *data table* con nombre “DT_ConstructionCost” y ruta en “/Content/StudentName/Placeables/Buildings” y que toma como referencia la estructura “S_ElementCost”. Es un dato de tipo “name” que se ha de corresponder exactamente con el nombre de la columna “Row Name” de esta *data table*, respetando incluso mayúsculas y minúsculas.
- en el caso de un edificio básico, de almacén o recolector, la **flag “producer_building_”** se mantendrá desactivada y sólo se activará en la jerarquía de herencia de “BP_ProducerBase” –activado por defecto en esta clase–.
- finalmente, se le indicará la cantidad de **población** que va a sumar el edificio – sea la cantidad que sea– y cuántos **trabajadores** va a necesitar, que en el caso de edificios básicos normalmente será cero, y en el de los productores una cantidad variable.



5. A continuación, el diseñador **ajustará algunos datos** de la nueva clase creada: dará valor a las dimensiones del “UBoxCollisionComponent” de la clase, asignará la geometría correspondiente al “UStaticMeshComponent”, ajustará incluso el escalado de la geometría –que no del componente– en caso de necesitarlo, así como colocará el “DoorPoint” en el lugar que le corresponda –este componente será la puerta del edificio–. Importante: nótese que, para un mapa demo en *grid* en



el que cada casilla ocupa unas dimensiones de "200.0f * 200.0f" en los ejes de la "X" e "Y", la "BoxExtent" de la cápsula no supera esas dimensiones en total para evitar que hayan edificios que se solapen –en el ejemplo es de "198.0f * 198.0f".



- Una vez ha hecho esto, el diseñador ha de ir a la *data table* de nombre "**DT_ConstructionCost**" que encontrará en la carpeta con ruta en "/Content/StudentName/Placeables/Buildings" y llenar la información relativa al **coste de construcción** del nuevo edificio –revisar que se ha modificado la estructura "S_ElementCost" conforme a los recursos de la plantilla–.

El nombre que le vayamos a poner a la edificación en la columna "Row Name" es importante porque se va a corresponder exactamente con el mismo nombre en otras *data table* como el que sigue en el siguiente punto. Por lo pronto, en esta *data table* se establece el coste de cada uno de los edificios planteados para cada uno de los recursos –materias y manufacturados– del juego:

Row Name	NeededWood	NeededGold	NeededIron	NeededAge	NeededMoney
1 House	{"Type": "EResourceType_Wood", "Cost": 0}	{"Type": "EResourceType_Iron", "Cost": 0}	{"Type": "EResourceType_Gold", "Cost": 0}	EAgeState_Wildcard	2.000000
2 LumberCamp	{"Type": "EResourceType_Wood", "Cost": 0}	{"Type": "EResourceType_Iron", "Cost": 0}	{"Type": "EResourceType_Gold", "Cost": 0}	EAgeState_Wildcard	20.000000
3 Sawmill	{"Type": "EResourceType_Wood", "Cost": 20}	{"Type": "EResourceType_Iron", "Cost": 20}	{"Type": "EResourceType_Gold", "Cost": 0}	EAgeState_Wildcard	10.000000
4 TownCenter	{"Type": "EResourceType_Wood", "Cost": 2}	{"Type": "EResourceType_Iron", "Cost": 2}	{"Type": "EResourceType_Gold", "Cost": 2}	EAgeState_Wildcard	4.000000

- Por ello también es recomendable, en este punto, llenar la *data table* de nombre "**DT_BuildingInfo**" y ruta en "/Content/StudentName/UI" con datos relacionados con la **información** que tiene que mostrarse por pantalla cuando pinchemos en un **botón que nos permita construir el nuevo edificio**. El dato que tenemos en la columna de nombre "RowName" ha de corresponderse exactamente con el "RowName" de la *data table* anterior –"DT_ConstructionCost"–.

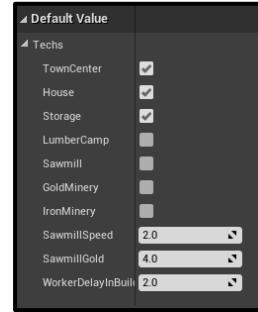
Row Name	BuildingType	StaticMesh	Name	Description	Image	BoxExtent	BuildingZone
1 House	EBuildingType_House	StaticMesh/Engine/BasicShapes/Cube/Cube'	House	House description because llore macu macu llore macu macu	Texture2D/Engine/EngineResources/ICON-Green/ICON-Green	(X: 99, "Y: 99, "Z: 64)	EZoneType_Wildcard
2 LumberCamp	EBuildingType_LumberCamp	StaticMesh/Engine/BasicShapes/Cone/Cone'	Lumber Camp	Lumber Camp description because llore macu macu llore macu macu	Texture2D/Engine/EngineResources/ICON-Red/ICON-Red	(X: 99, "Y: 99, "Z: 64)	EZoneType_Forest
3 Sawmill	EBuildingType_Sawmill	StaticMesh/Engine/BasicShapes/Sphere/Sphere'	Sawmill	Sawmill description... because... llore macu macu llore macu macu	Texture2D/Engine/EngineMaterials/BlendFunc_Def	(X: 99, "Y: 99, "Z: 64)	EZoneType_Wildcard
4 TownCenter	EBuildingType_TownCenter	StaticMesh/Engine/BasicShapes/Cube/Cube'	TownCenter	This is the Town Center bla bla	Texture2D/Engine/Engine_ML_Shaders/Textures/Bokeh/Bokeh'	(X: 199, "Y: 199, "Z: 64)	EZoneType_Wildcard

- A continuación, hay que añadir en la estructura "**S_TechCollection**" que encontramos en "/Content/StudentName/UI/TechnologyTree/TechTreeTasks" el **booleano para la tecnología** relacionada con el nuevo tipo de edificio:

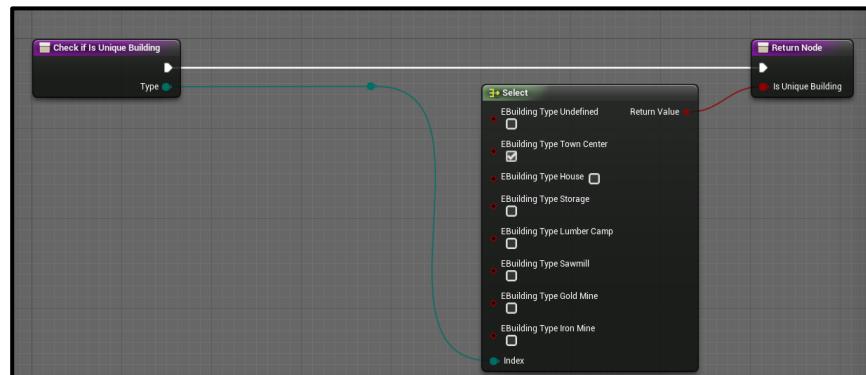




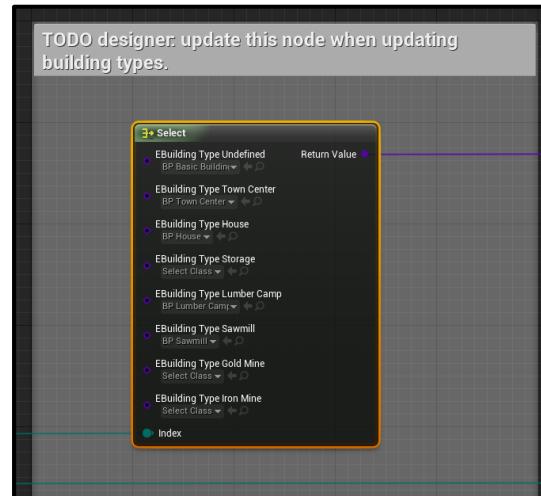
9. Si queremos que esta tecnología ya comience descubierta y, por lo tanto, se pueda comenzar a construir un edificio desde el inicio y sean visibles los botones relacionados con él, lo marcamos en la variable correspondiente del dato “**techs_**” de “**BP_GameState**”:



10. Una vez hecho esto, toca **reconectar funcionalidad** en el método “**CheckIfBuildingTechIsDiscovered**” de “**BP_GameState**” para que el juego conozca si ese edificio se puede construir porque la tecnología se ha descubierto, así como **marcar si es un edificio único** en “**CheckIfsUniqueBuilding**”. En caso de ser un edificio único, se recomienda consultar detalles más específicos en el apartado relacionado con los edificios únicos, así como consultar también el apartado relacionado con el árbol de tecnologías.



11. Para finalizar, hay que **actualizar la clase a instanciar** en el “Select” que encontramos en el **método “SpawnBuilding”** de la clase “**BP_Pawn**” en base al tipo de edificio:





12. En los siguientes subepígrafes, lo explicado en este apartado se va a ampliar en casos concretos y particulares para ofrecer al diseñador una perspectiva más específica de cada tipo.

5.8.1. Especificidades en edificios básicos

En primer lugar, a la hora de heredar de un edificio básico, si acaso, hay que tener en cuenta que, en la nueva clase derivada, seguramente hay que **sobreescibir** el método de la clase base “**ExecuteActionWhenBuilt**” y el “**EventDestroyed**”. En principio, para estos tipos de edificios, no habría necesidad de sobreescibir ningún método más.

El diseñador cuenta en la plantilla con un ejemplo de este edificio en la clase “BP_House” con ruta en “/Content/StudentName/Placeables/Buildings/Basics”.

5.8.2. Especificidades en edificios recolectores

En segundo lugar, a la hora de heredar de un edificio recolector de materias primas, y a diferencia de lo que hacíamos cuando heredábamos de “BP_BasicBuilding”, en principio no se atisba necesario que se reescriba ningún método pero sí es recomendable **revisar la funcionalidad escriptada en la clase base “BP_CollectorBase”**.

Esta recomendación viene motivada por si el diseñador tuviera que sobreescibir en la nueva clase la funcionalidad algún método o evento: “ExecuteActionWhenBuilt”, “EventDestroyed”, “UpdateWorkerMultiplier”, “EventBeginPlay”, “ChangeActiveState”…

Destacar en último lugar que el diseñador dispone de un ejemplo de edificio recolector de materias primas en la clase “BP_LumberCamp”, disponible para consulta o modificación en “/Content/StudentName/Placeables/Buildings/Economy/Collectors”.

5.8.3. Especificidades en edificios productores

Por otra parte, a la hora de heredar de un edificio productor, en este caso en concreto el diseñador tal vez tenga que **reescribir con casi total seguridad algunos métodos** en la clase derivada como puedan ser “ExecuteActionWhenBuilt”, “EventDestroyed”, “ChangeActiveState”, “ManualActivationTransporter”, “ManualActivationWidget”, “ManualDeactivationWidget”, “EventBeginPlay”…

Resaltar que el desarrollador dispone de un ejemplo de edificio productor en la clase “BP_Sawmill”, disponible para consulta o modificación en la ruta “/Content/StudentName/Placeables/Buildings/Economy/Collectors”. En este caso, se trata de una serrería que convierte madera en monedas.

5.8.4. Especificidades en edificios almacenes

Finalmente, a la hora de heredar de un edificio almacén, hay que tener en cuenta que de nuevo **puede ser necesario reescribir en la nueva clase derivada algunos métodos** según necesidades propias del desarrollo como puedan ser los métodos o eventos “ExecuteActionWhenBuilt”, “EventDestroyed”, “ChangeActiveState”, “ManualActivationTransporter”, “ManualActivationWidget”, “ManualDeactivationWidget”, “EventBeginPlay”…

Comentar que el diseñador dispone de un par de ejemplos en las clases “BP_Storage” y “BP_TownCenter”, ambas clases que se encuentran en la carpeta con ruta en “/Content/StudentName/Placeables/Buildings/Storageables”.

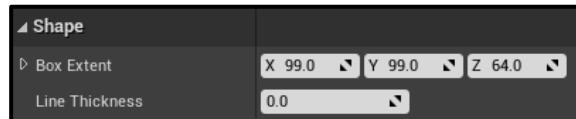


5.9. Instanciar edificios multicasillas

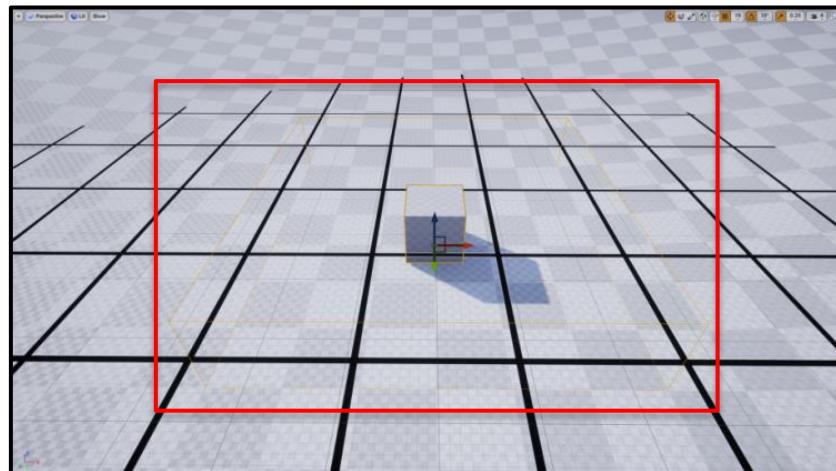
En este punto se va a abordar la problemática que sucede a la hora de determinar las dimensiones del “BoxExtent” del “UBoxCollisionComponent” que encontramos en todo edificio, heredado desde la clase base “BP_BasicBuilding”, y que cobra especial relevancia en un **mundo de juego en grid** –en uno libre daría un poco más igual–.

Si el diseñador desea crear un edificio que ocupe **una única casilla**, lo ideal es que las **dimensiones de su “BoxExtent” no superen las dimensiones de esa casilla** sobre el mundo de juego –revisar el apartado sobre la configuración del “GridWorld” ya que hay un subpunto que versa justamente sobre esto–.

En los niveles de ejemplo se puede observar cómo, para mundos de juego en forma de *grid* en los que cada casilla ocupa unas dimensiones de “200.0f * 200.0f” en los ejes de la “X” e “Y”, clases de edificios ya creadas por defecto como “BP_House” tienen el “BoxExtent” de su “UBoxCollisionComponent” un pelín por debajo del tamaño de la casilla –“198.0f * 198.0f” en este ejemplo–:



No obstante, si el diseñador pretende crear un **edificio que ocupe varias casillas**, es tan sencillo como que coja y le dé unas **dimensiones adecuadas al “BoxExtent”** del “UBoxCollisionComponent” del edificio en cuestión, así como jugar con el propio “Transform” del “UStaticMeshComponent” del edificio. Es algo que puede verse en, por ejemplo, la clase “BP_TownCenter” que encontramos dentro de la carpeta con ruta en “/Content/StudentName/Placeables/Buildings/Storageables” y del que puede verse un actor instanciado en los niveles:



5.10. Instanciar edificios en zonas

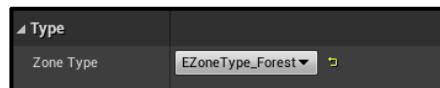
Si el diseñador pretende crear **zonas especiales de instanciación**, al margen de cómo decida adornarlo visualmente, a nivel interno de código ha de trabajar con la clase **“BP_ZoneTrigger”** que encontrará en la jerarquía de carpetas con ruta en “/Content/StudentName/Placeables/Zones”.



Es importante que, antes de lanzarse a instanciar *triggers* de esta clase sobre el nivel, el diseñador añada los **nuevos tipos de zonas** en el “enum” de nombre “**EZoneType**” que encuentra en la misma jerarquía de carpetas.



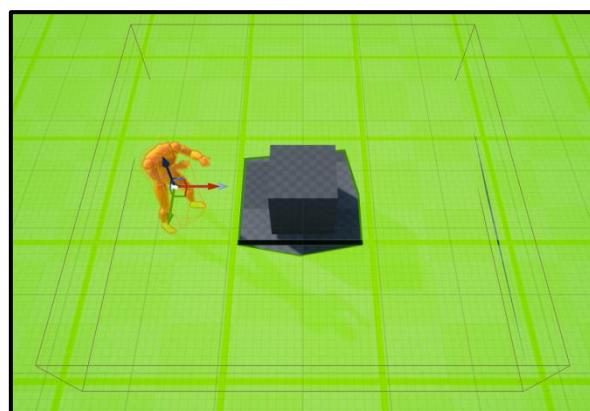
A partir de ahí, no se atisba necesario en este caso en concreto reimplementar ninguna funcionalidad en particular de este *trigger* sino que es cuestión de utilizarlo sobre el nivel en aquellas partes del mapa de juego en las que se quiera **delimitar una zona de instalación**, darle el tipo correspondiente y asegurarnos en los edificios que queramos instanciar únicamente en la zona delimitada por el *trigger* que su tipo de zona es el mismo.



Destacar, en último lugar, que no hay que instanciar ningún *trigger* en concreto si queremos que un edificio pueda ser construido en cualquier lugar, simplemente decirle a ese tipo de edificio que su tipo de zona es “**wildcard**” porque eso significará que **puede ser instanciado en cualquier punto** del mapa.

5.11. Edificios y “Navigation Mesh”

Independientemente del tamaño que tengan las “StaticMesh” que se emplearán en los edificios, es importante **dejar espacio suficiente entre la geometría y los límites del “UBoxCollisionComponent”** que encontramos dentro de cada edificio para que los agentes de inteligencia artificial puedan circular libremente por él.

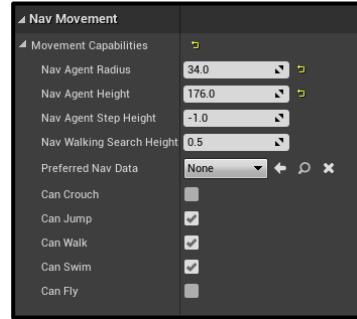


5.12. Interacción entre sí de los agentes de IA

Relacionado estrechamente con el apartado anterior, es importante destacar también que los agentes de inteligencia artificial incluidos en la plantilla están manejados mediante sendos **controladores** que heredan ambos de “ADetourAICrowdController” **para el manejo de gestión de multitudes**. Para que este gestor de masas funcione



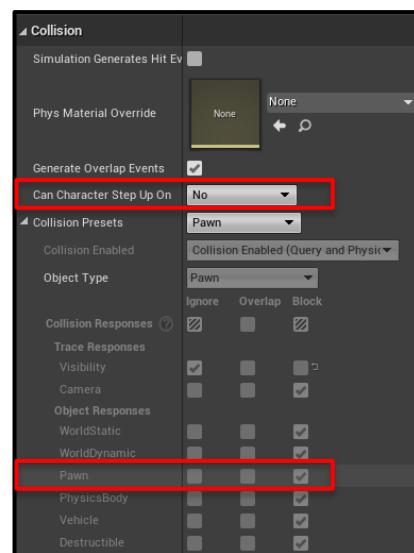
bien, es importante, en primer lugar, **ajustar los valores de navegación** sobre el “Nav Mesh” de los “**Movement Capabilities**” del “UCharacterMovementComponent” de cada agente acorde a las dimensiones de la cápsula de cada uno:



Es importante recordar, en segundo lugar, que **el gestor de masas se ajusta en “Project Settings”** para, por ejemplo, modificar el número máximo de agentes que va a poder controlar para evitar que traten de chocarse.



En último lugar, pero no por ello menos importante, a destacar que ahora mismo los agentes de inteligencia artificial colisionan entre ellos pero es algo que se puede personalizar en los ajustes de la sección “Collision” de la clase base “BP_NPC” para evitar que “Pawn” bloquee a “Pawn”. En caso de modificar la clase base y no las derivadas, es recomendable comprobar en las derivadas que los cambios se han realizado y heredado correctamente.



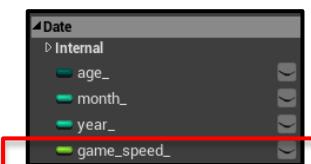


5.13. Configurar año, edad y velocidad *in game*

Es en la clase “BP_GameState” del *framework* principal de clases en las que el diseñador puede determinar el mes, año y edad en la que va a comenzar el juego. La plantilla viene con unas edades por defecto que el diseñador puede modificar en la medida de sus necesidades, para ello, ha de hacerlo en el “**enum**” de nombre “**EAgeState**” que encontrará en la jerarquía “/Content/StudentName/Placeables”:



En definitiva, el **epígrafe “Date” en “BP_GameState”** donde es posible modificar mes, año y edad iniciales. Adicionalmente, hay un atributo de nombre “game_speed_” que es el que simboliza cuántos segundos de la vida real se necesitan para avanzar un mes en el juego.



5.14. Misiones de un nivel

En el momento de la entrega de la plantilla existe un único modo de juego cuya única función es aglutinar las clases básicas del *framework* principal que se utilizarán durante la partida: no hay ninguna funcionalidad programada de antemano en la **clase “BP_GameModeBase”** que se está utilizando por defecto. No obstante, es en esta clase en la que el diseñador programará toda aquella funcionalidad relativa con las **condiciones de victoria y / o derrota** del jugador para el nivel por defecto de la práctica.

5.15. Campañas de varios niveles

En el supuesto de que el diseñador quisiera crear una campaña formada por más de un nivel consecutivos, además de crear un nuevo nivel y configurarlo previamente en tiempo de edición, ha de asignar un **nuevo modo de juego** que reutilice la clase “BP_GameState” así como el resto de clases del *framework* principal. La razón de crear un nuevo modo de juego es porque es la clase en la que el diseñador establecerá, potencialmente, las condiciones de victoria y / o derrota del jugador.

Prácticamente toda la **funcionalidad programada para un nivel debería de servir para crear nuevos niveles** en los que establecer un nuevo modo de juego con sus respectivas condiciones de victoria y / o derrota, crear si acaso un árbol de tecnologías un pelín diferente y diseñar mapas con una predisposición distinta de los elementos.

5.16. Configurar el árbol de tecnologías

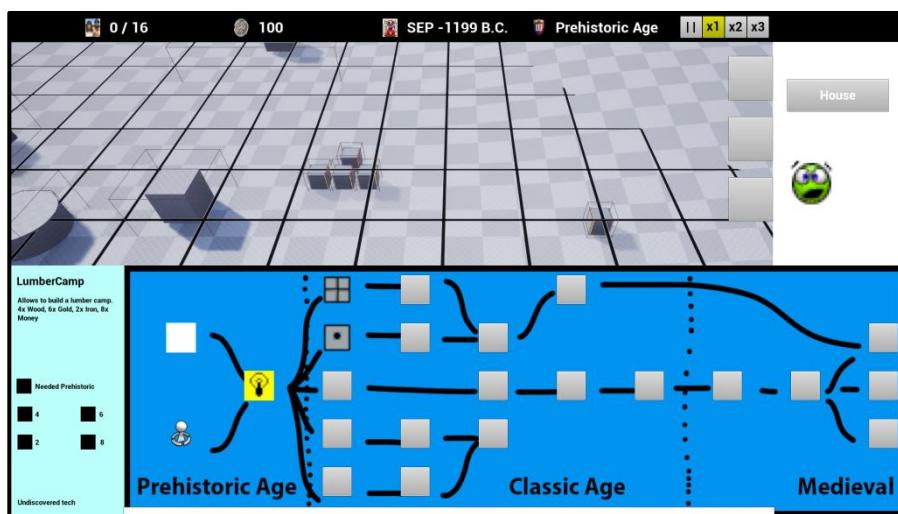
El árbol de tecnologías supone uno de los sistemas *core* de la plantilla y se basa, por una parte, en el manejo de una serie de botones predisuestos de forma jerarquía y dependiente entre ellos; y, por otra parte, en unos actores tareas que el desarrollador ha de crear y que se ejecutan al terminar de completar de investigar una tecnología. El



objetivo del sistema es resultar lo más cómodo posible al diseñador el hecho de configurar un árbol de tecnologías:

1. En primer lugar, el diseñador debe de tener en cuenta que el árbol de tecnologías actualmente se muestra en la parte inferior de la pantalla al pulsar el input correspondiente, por lo que permite continuar viendo el mundo de juego y, por lo tanto, en el momento en el que se ha entregado la plantilla este no detiene la ejecución del juego. El diseñador puede **remodelar estéticamente el árbol de tecnologías** para colocarlo en el lugar de la pantalla que considere y modificar a su gusto el hecho de que pause o no el juego.

Visualmente, ahora mismo, el árbol de tecnologías luce de la siguiente forma con el juego en ejecución:



2. En segundo lugar, el diseñador ha de **configurar el árbol** que encontrará en el *widget* de nombre “WB_TechTree” **con widgets “WB_TechTreeButton”**: cada botón simbolizará una nueva tecnología que el jugador podrá descubrir. Estos *widgets* se encuentran en la jerarquía de carpetas con ruta en “/Content/StudentName/UI/TechnologyTree”. Cada uno de los nuevos botones de tecnología que el diseñador cree se corresponderá con una tecnología diferente y todos sin excepción se meterán dentro del “UniformGridPanel” que encontramos dentro de la clase de *widget* de nombre “WB_TechTree”:





3. A la hora de diseñar visualmente la composición del estilo del árbol de tecnologías, que es similar al que nos encontramos en juegos como la saga “Civilization”, para mostrar al jugador la **relación de parentesco entre tecnologías y en qué época** se puede desbloquear cada una, lo recomendable es que el diseñador cree una **textura** tan larga como sea el “Uniform Grid Panel” –mismas dimensiones– y que enseñe claramente esa información como sucede en la textura de ejemplo de la plantilla.
4. El diseñador puede configurar tantas tecnologías como necesite para su idea de juego, pero **toda tecnología que se cree tiene que estar en** la estructura de datos de nombre “**S_TechCollection**”, con ruta en la carpeta del proyecto “/Content/StudentName/UI/TechnologyTree/TechTreeTasks” –revisar el apartado de la documentación “Modificación de las estructuras de datos”–. La única **excepción**, si acaso, son las **tecnologías para avanzar de edad**, el resto de información estará recogida en esta estructura de datos:

Structure	
	New Variable
Tooltip	
► TownCenter	Boolean
► House	Boolean
► Storage	Boolean
► LumberCamp	Boolean
► Sawmill	Boolean
► GoldMinery	Boolean
► IronMinery	Boolean
► SawmillSpeed	Float
► SawmillGold	Float
► WorkerDelayInBuilding	Float

5. De esta estructura de datos existe un atributo en la clase “**BP_GameState**” que se emplea en el método “**CheckIfBuildingTechIsDiscovered**”. Como se puede intuir, las tecnologías sirven al jugador para desbloquear nuevos edificios, para avanzar de época y para potenciar ciertas actividades en su aldea. Si quisieramos que alguna tecnología comenzara **por defecto descubierta**, es tan sencillo como darle ese valor por defecto a la **variable “techs_”** de la clase “**BP_GameState**”:

Default Value	
► Techs	
TownCenter	<input checked="" type="checkbox"/>
House	<input checked="" type="checkbox"/>
Storage	<input checked="" type="checkbox"/>
LumberCamp	<input type="checkbox"/>
Sawmill	<input type="checkbox"/>
GoldMinery	<input type="checkbox"/>
IronMinery	<input type="checkbox"/>
SawmillSpeed	2.0
SawmillGold	4.0
WorkerDelayInBuilding	2.0

6. A continuación, en este punto es recomendable **comprobar que hay información ya insertada en el sistema sobre edificios** –revisar el apartado “Añadir nuevos tipos de edificios”– y asegurarse también de lo siguiente:

- a. El diseñador ha de ir a la *data table* de nombre “**DT_BuildingInfo**” con ruta en “/Content/StudentName/UI” y revisar en este punto que todos los



edificios planeados han sido añadidos. Puede añadir nuevos mediante el botón de “+”. Asegurarse que, en el caso de aquellas tecnologías que van a descubrir un nuevo edificio, el “RowName” de esta tabla...:

Row Name	BuildingType	StaticMesh
1 House	EBuildingType_House	StaticMesh'/Engine/Bas
2 LumberCamp	EBuildingType_LumberCamp	StaticMesh'/Engine/Bas
3 Sawmill	EBuildingType_Sawmill	StaticMesh'/Engine/Bas
4 TownCenter	EBuildingType_TownCenter	StaticMesh'/Engine/Bas

b. ... se corresponda con el nombre de la *data table* de nombre “DT_ConstructionCost”.

Row Name	NeededWood
1 House	{"Type": "EResourceType_Wood", "Cost": 0}
2 LumberCamp	{"Type": "EResourceType_Wood", "Cost": 0}
3 Sawmill	{"Type": "EResourceType_Wood", "Cost": 20}
4 TownCenter	{"Type": "EResourceType_Wood", "Cost": 2}

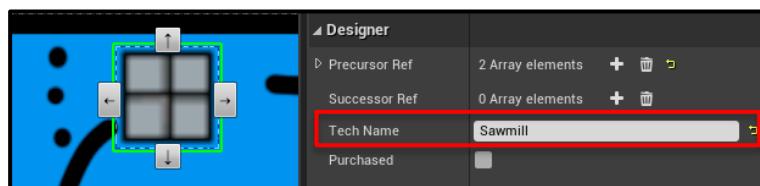
7. Estas dos últimas *data table* no guardan relación directa de “RowName” con la *data table* “DT_TechInfo” pese a que pueda parecer lo contrario, pero sí indirecta en tanto que es en esta última *data table* en la que vamos a configurar las distintas **tecnologías que se pueden investigar**: avanzar de edad, mejoras y permitir construir nuevos edificios.

- “**Name**” es el nombre que aparecerá en la descripción de la tecnología que encontramos al colocar el ratón sobre su respectivo botón en el *widget* de nombre “WB_TechTreeResourcesInfo” que hay incluido en “WB_TechTree”.
- “**Description**” es la descripción de la tecnología, también aparecerá en el mismo *subwidget*.

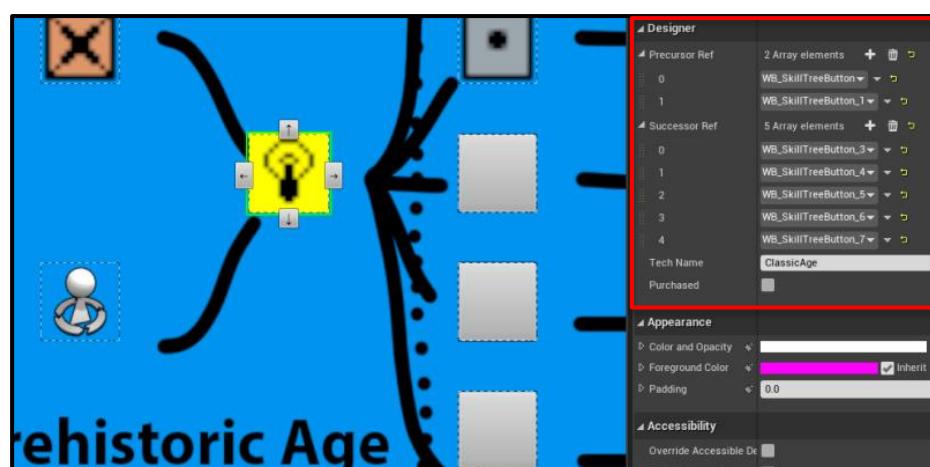


- “**Image**” es la imagen que ilustrará el botón de la clase “WB_TechTreeButton”.
- “**CompletedImage**” es la imagen que mostrar en el botón de la clase “WB_TechTreeButton” cuando la tecnología termine de investigarse.
- “**TimeToComplete**” es el lapso de tiempo en segundos que va a tardar el juego en completar de investigar la tecnología.
- “**Cost**” es el coste de investigación de la tecnología, no hay que confundirlo con el coste de construcción de un edificio en el caso de que la tecnología sirviera para investigar un nuevo edificio.
- “**TaskClass**” es la clase de la tarea asociada a la tecnología, que va a resultar efectiva cuando termine de completarse la investigación de la nueva tecnología. Al final de este epígrafe se detallará cómo crear nuevas tareas.

8. Lo siguiente es ir a cada uno de los “WB_TechTreeButton” instanciados manualmente dentro del “WB_TechTree” y, en el **atributo público “tech_name”** que encontramos en los “Details” de cada botón, vamos escribiendo las distintas “Row Name” del *data table* “DT_TechInfo”.



9. Para terminar de configurar las distintas tecnologías del árbol, hay que decirle a cada una de las tecnologías quiénes van a ser las **tecnologías precursoras o sucesoras**, en caso de que la tecnología en cuestión tenga. Además, podemos hacer que una tecnología comience **comprada por defecto mediante la flag “purchased_”** para que se ejecute nada más comenzar la partida. El siguiente ejemplo, extraído directamente de la plantilla, ilustra el parentesco de la tecnología para avanzar a la “Edad Clásica”:

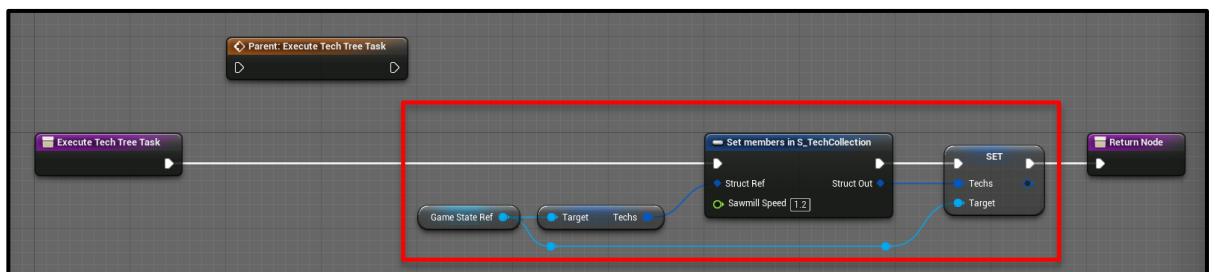


10. Finalmente, hay que **crear las diversas tareas** que se van a ejecutar cuando una tecnología termina de ser investigada. El paradigma que se ha seguido para crear estas tareas es el de una actores sin “Tick” que hereden de una clase de “Blueprint” de nombre “BP_TechTreeTask” que encontramos dentro de la jerarquía de carpetas de “/Content/StudentName/UI/TechTree/TechTreeTasks”.



11. Esta clase base contiene un **método de nombre “ExecuteTechTreeTask”** que es el que hay que sobreescribir en cada una de las clases derivadas. La plantilla incluye ejemplos para investigar nuevos edificios, para realizar mejoras *in game* y también para avanzar de época. En el caso de las mejoras *in game*, es importante que luego se empleen en la funcionalidad propia de cada edificio como es el caso de la variable “*sawmill_speed_*” que se encuentra dentro del atributo “*techs_*” de la clase “*BP_GameState*”, valor empleado en la clase de edificio “*BP_Sawmill*” que viene de demostración junto a la plantilla y que es el tiempo que tarda en producir este edificio.

El siguiente ejemplo muestra el método “*ExecuteTechTreeTask*” que encontramos dentro de la clase “*BP_SawmillSpeed*” y que juega justamente con cambiarle el valor a la variable “*sawmill_speed*” del atributo “*techs_*” de la clase “*BP_GameState*”:

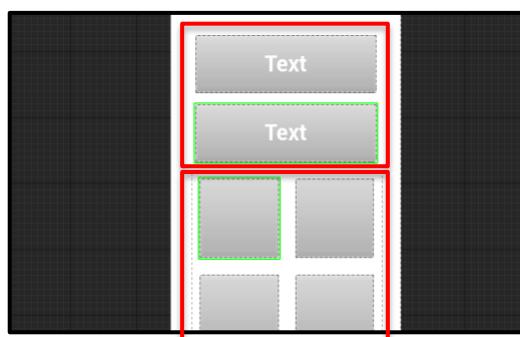


12. En caso de que la tecnología descubierta sea para construir un nuevo tipo de edificio y sea un edificio de construcción única, se recomienda al diseñador consultar el punto de la documentación relacionado con crear edificios únicos, así como también consultar la documentación relacionada con crear nuevos tipos de edificios.

5.17. Interfaz de construcción

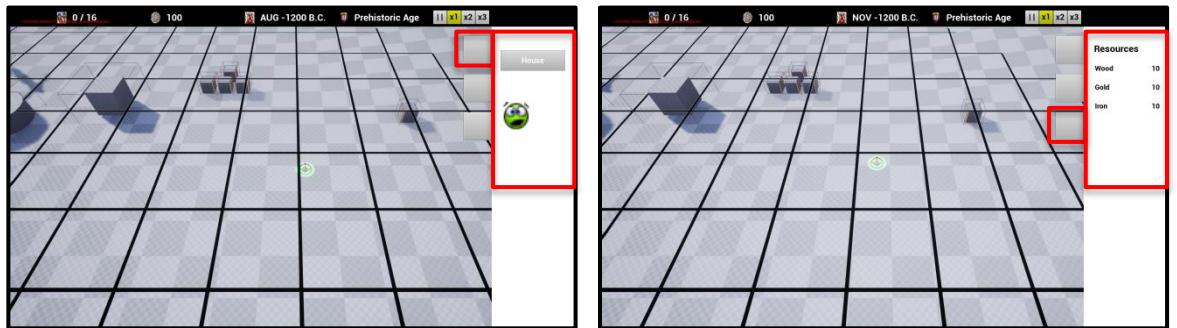
La creación de las páginas de construcción sigue un sistema modular en el que el diseñador va a utilizar indistintamente un par de clases botones de nombre “*WB_ConstructionButtonImage*” y “*WB_ConstructionButtonText*” englobadas dentro de otros objetos que van a actuar como páginas de interfaz de usuario.

1. Para empezar, el diseñador tiene **total libertad de diseño** a la hora de utilizar únicamente botones de imagen de la clase “*WB_ConstructionButtonImage*”, de emplear únicamente botones de texto de la clase “*WB_ConstructionButtonText*” o de utilizar ambos tipos de forma simultánea en su juego: ambos hacen exactamente lo mismo y tienen toda la funcionalidad ya programada, sólo que el contenedor es diferente.





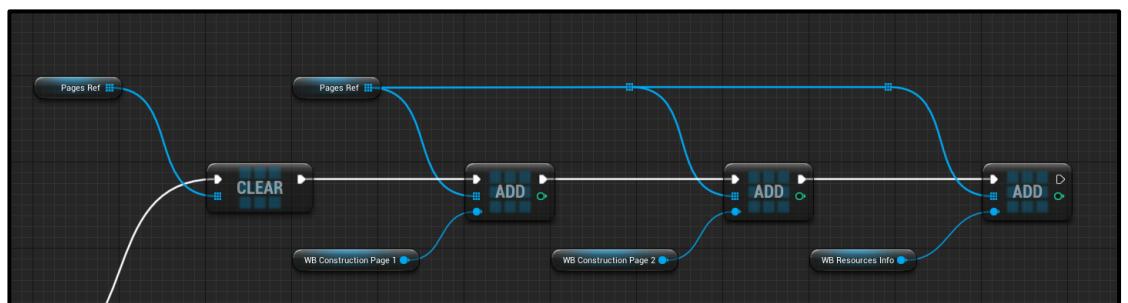
2. El diseñador puede **configurar la interfaz de construcción** a su gusto: en la plantilla hay, por defecto, una serie de pestañas laterales con botones de los edificios disponibles para construcción y los recursos que posee el jugador:



3. Estas **interfaces visuales** que vienen por defecto con la plantilla se encuentran **dentro del widget "WB_ConstructionInterface"** y con ruta en la carpeta "/Content/StudentName/UI/ConstructionPages". Es esta clase de *widget* la que el desarrollador ha de modificar tanto para añadir funcionalidad como para predisponer los elementos de otra forma. En el momento de la entrega de la plantilla, este *widget* contiene otros dos *subwidgets*:

- un par de *widgets* que heredan de la clase "WB_ConstructionPageBase" con ruta en la carpeta "/Content/StudentName/UI/ConstructionPages" y con una serie de botones de construcción a modo de demo. En uno de ellos, además, se utiliza la clase de *widget* "WB_WidgetInfo" con ruta en la jerarquía de carpetas "/Content/StudentName/UI/Info".
- un tercer *widget* que, si bien también hereda de "WB_ConstructionPageBase", sólo contiene un *widget* de la clase "WB_ResourcesInfo" que se ha aprovechado a colocar como *subwidget* para informar al jugador de los recursos que posee en el juego –*widget* que ya se ha especificado anteriormente que hay que modificar y adaptar conforme el diseñador cree nuevos recursos en su juego–.

4. A la hora de crear nuevas páginas de construcción e insertarlas dentro del *widget* "WB_ConstructionInterface", el diseñador ha de **heredar de la clase *widget* "WB_ConstructionPageBase"**. En la plantilla hay tres herencias de ejemplo que utilizan ambos tipos de botones pero, como se ha explicado anteriormente, no es necesario ni mucho menos emplear ambos simultáneamente sino el que sea más del agrado del diseñador. Eso sí: **toda página de construcción creada ha de ser dentro de "WB_ConstructionInterface"** y, en el "Event Pre Construct" de esta, se ha de añadir el nuevo *widget* en el array "pages_ref_":



5. Una vez colocados todos los botones, sean de texto o imagen, sobre las páginas que el diseñador ha creado, es importante revisar que el nuevo edificio ha sido creado con éxito en la plantilla, por lo que se recomienda **revisar el apartado relativo a añadir nuevos tipos edificios**.

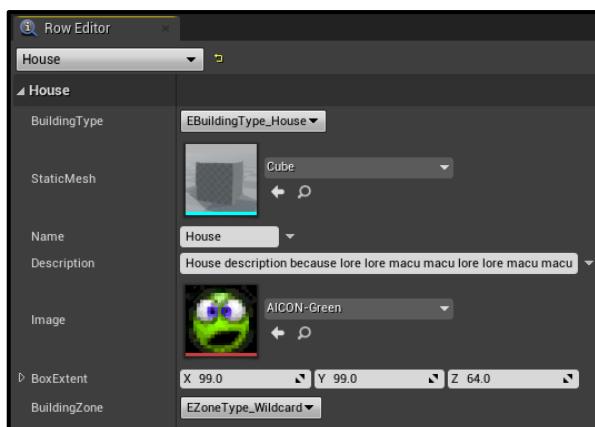


6. Una vez asegurado esto, hay que ir a la *data table* “DT_BuildingInfo” y, si no se había hecho previamente, hay que llenar esa tabla con información de aquellos edificios que el jugador tenga opción de construir en su juego.

Como se ha explicado en epígrafes anteriores, es importante que el nombre que pongamos en la columna “RowName” se corresponda exactamente con el “RowName” de la *data table* de nombre “DT_ConstructionCost”. Para cada entrada, al margen del “RowName”, los campos a llenar son los siguientes:

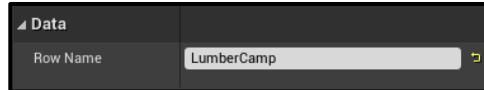
- “BuildingType” es el tipo del edificio a instanciar y ha de corresponderse con su tipo real en la clase correspondiente. Este dato es importante porque, si no se pone bien, puede que se instancie un edificio totalmente diferente al que debería.
- “StaticMesh” es exactamente la misma geometría que encontraremos en la clase de edificio asociada con este botón. Esta geometría sirve para comunicársela a la instancia de la clase “BP_PrefabBuilding” asociada con la instancia de la clase “BP_Pawn” que controla el jugador y poder previsualizar antes de colocar un edificio cómo este ha de verse.
- “Name” y “description” son datos que la plantilla ya comunica con aquellos *widgets* de la clase “WB_WidgetInfo” que tengamos creados. Adicionalmente, “name” sirve para darle texto a los botones de texto.
- “Image” sirve para darle imagen a los botones de imagen.
- “BoxExtent” es un atributo empleado, al igual que el atributo “StaticMesh”, para darle a la instancia de la clase “BP_PrefabBuilding” que tenemos por defecto en el nivel información sobre los bordes del edificio para saber dónde podemos instanciarlo. Además, y al igual que con la “StaticMesh”, de nuevo nos encontramos que este dato se ha de corresponder con los *bounds* de la “BoxCollisionComponent” de la clase del edificio relacionada con el botón.
- “BuildingZone”, finalmente, sirve para determinar la zona específica en la que se va a poder instanciar el edificio asociado con el botón, y ha de corresponderse con el dato del edificio.

La siguiente imagen de ejemplo ilustra esta información para la “RowName” de nombre “House”, asociada a su vez con el dato “EBuildingType_House”, y a su vez relacionada con la clase “BP_House”. Todos son datos importantes para, en definitiva, que el usuario previsualice cómo va a lucir un edificio sobre el mundo, cuánto le va a costar construirlo –“DT_ConstructionCost”– y pueda instanciarlo efectivamente.





- Finalmente, cuando la *data table* “DT_BuildingInfo” se ha rellenado correctamente, hay que ir **botón a botón** en las páginas de construcción y **asignarle el nombre de la “RowName” correspondiente**.

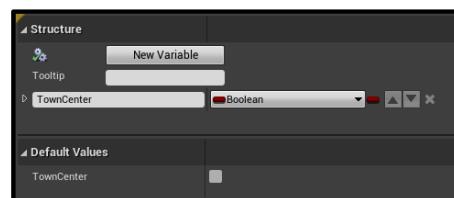


- A destacar que, cuando el jugador ya ha instanciado sobre el mundo de juego un edificio único o no ha descubierto todavía la tecnología para construir un edificio cualquiera, el botón asociado a ese tipo de edificio no va a aparecer, característica también programada por defecto. Los tipos de edificios descubiertos para poder ser construidos se guardan en la variable “techs_” de “BP_GameState” y el diseñador puede hacer, por defecto, que un edificio ya haya sido investigado y el jugador pueda comenzar a construirlo desde el inicio. Para más información, consultar los apartados relativos a construir edificios únicos y a añadir nuevos tipos de edificios.

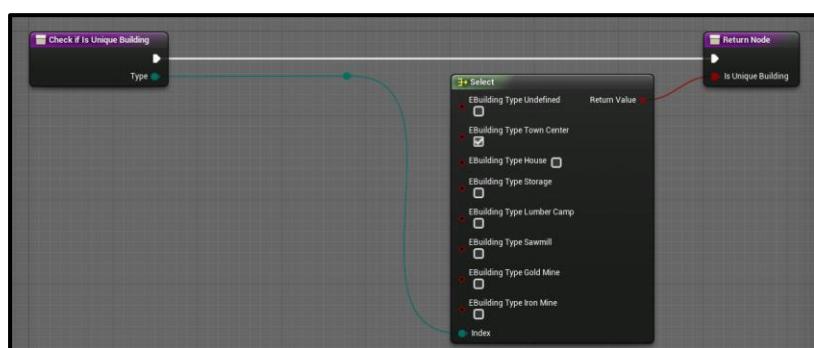
5.18. Edificios únicos

Los edificios únicos son aquellos de los que sólo se pueden construir simultáneamente una única instancia en todo el juego. Es una característica que permite la plantilla y para la que el diseñador ha de realizar los siguientes pasos:

- En primer lugar, hay que **añadir la flag correspondiente en** la estructura de datos **“S_UniqueBuilding”** con ruta en “/Content/StudentName/Placeables/Buildings” y utilizada en la clase del *framework* “BP_GameState”, concretamente en el atributo “unique_buildings_state_” –revisar el apartado “Modificación de las estructuras de datos”–.

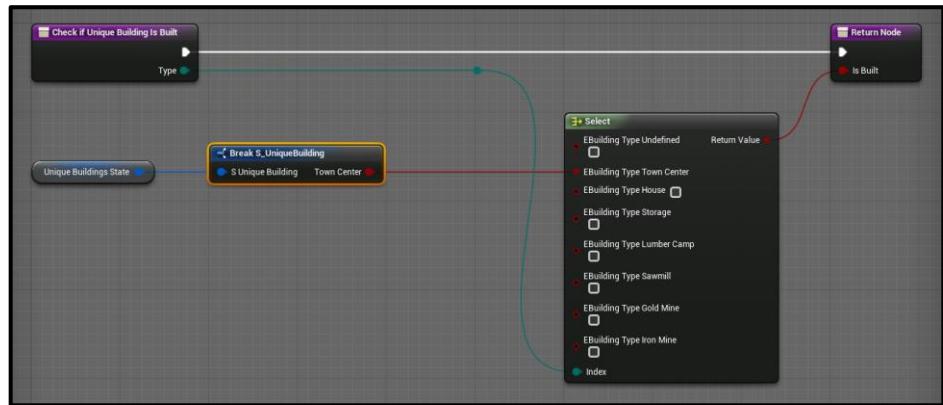


- En la **clase “BP_GameState”** hay que revisar la funcionalidad programada en un par de funciones: por una parte, hay que actualizar el estado del “Select” mostrado en la siguiente captura en el **método “CheckIfIsUniqueBuilding”** en el que, básicamente, el desarrollador ticará qué edificios son únicos y cuáles no –se decidió meterlo en este sitio y no a nivel de la clase de edificio porque es un estado global del juego y, para consultararlo, no debería de haber ninguna instancia de un edificio creado en el mundo de juego–; ...

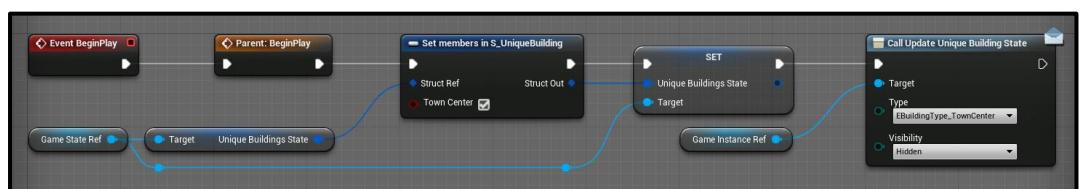




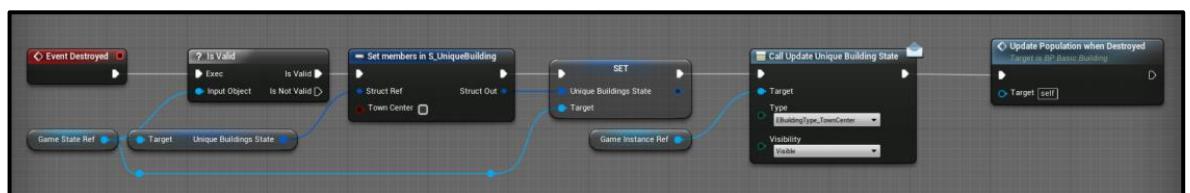
3. ...y, por otra parte, hay que revisar el **método “CheckIfUniqueBuildingIsBuilt”** para leer el valor de todas las *flags* que hayamos metido dentro de la estructura “S_UniqueBuilding” y que, como se ha explicado líneas arriba, sirve para actualizar *in game* qué edificios únicos se han construido, de modo que esta función lee los datos de la variable “unique_buildings_state_” y hay que reconectar las *flags* que el diseñador cree con su respectivo tipo en el “Select”.



4. La plantilla ya incluye funcionalidad programada en los botones para que, cuando se elimina un edificio y este es único, aquellos botones relacionados con ese edificio único vuelvan a ser interactuables por el usuario. No obstante, esta funcionalidad responde a una **llamada al evento “UpdateUniqueBuildingState”** de la clase “BP_GameInstance”, por lo que el desarrollador ha de incluir esta llamada en los puntos del código que se especifican en el siguiente punto.
5. Finalmente, a destacar que sólo quedaría crear la nueva clase de edificio –se sugiere revisar el punto relativo a añadir nuevos tipos de edificios– y, aparte de programar la funcionalidad que sea sobrecargando los métodos o eventos que sean, hay que **actualizar en el propio evento “BeginPlay”** el estado de construcción del nuevo edificio único mediante **la flag correspondiente del atributo “unique_buildings_state”**. A continuación, sólo quedaría realizar una llamada al evento “UpdateUniqueBuildingState” mentado en el punto anterior para que los botones de construcción relacionados con este edificio único se se invisibilicen. Del mismo modo, en el “EventDestroyed” habría que realizar lo mismo pero en sentido inverso.
6. “BP_TownCenter” de “/Content/StudentName/Placeables/Buildings/Storageables” incluye funcionalidad a modo de ejemplo para mostrar al diseñador cómo poder informar al sistema de que un edificio único se ha construido...



... o, por el contrario, se ha destruido:

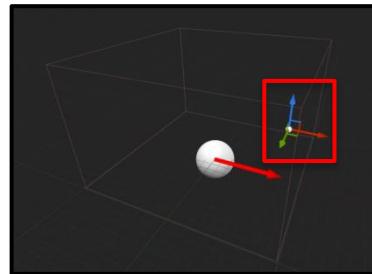




5.19. Door point

La clase base “BP_BasicBuilding” incluye un “**SceneComponent**” de nombre “**DoorPoint**” que es el que utilizan los agentes de inteligencia artificial para regresar a su respectivo edificio y, concretamente, a ese punto, como si de una puerta de entrada al edificio se correspondiera.

Como se puede apreciar en la siguiente imagen, este componente se ha de colocar individualmente por edificio y se ha de procurar, siempre, que se encuentre **dentro del “BoxCollisionComponent”** del edificio para que no invada el espacio de otros. Además, hay que tener en cuenta el “acceptance_radius_” de los nodos “MoveTo” que encontramos dentro de los “BehaviorTree” de nuestros agentes de inteligencia artificial.



A modo de último apunte, es un **componente hijo del “StaticMeshComponent”** que encontramos en estas clases porque, en el supuesto de que el usuario gire la geometría del edificio para instanciarlo un poco rotado, la “puerta” debería de rotar con el edificio. Es responsabilidad del diseñador el que este punto, gire como gire la geometría, siempre se va a mantener dentro de los límites del “BoxCollisionComponent” y así ha de ser. Además, el diseñador puede decidir prescindir de este input simplemente desconectándolo en la llamada en la clase “BP_PlayerController”.

5.20. Impedir instanciar un edificio

La plantilla incluye algunos mecanismos naturales para impedir al jugador que instancie un edificio en determinadas circunstancias. En primer lugar, **un edificio no va a poder instanciarse nunca si su “BoxCollisionComponent” se solapa con la de otro edificio**, por lo que es responsabilidad del diseñador ajustar los tamaños de las distintas cajas de colisión.

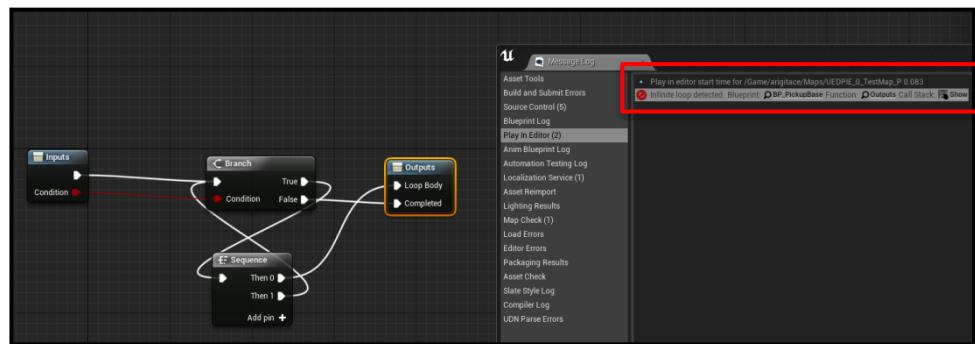
En segundo lugar, **si el jugador no dispone de los recursos suficientes** para crear un edificio, este no se va a construir, por lo que de nuevo vuelve a ser responsabilidad del diseñador tratar de balancear los sumideros de los recursos del juego. En tercer lugar, el **atributo “max_angle_spawn_”** de la clase “BP_GridWorld” indica el **ángulo máximo** al que se va a poder instanciar un edificio.

Adicionalmente, la plantilla incluye la **clase “BP_ZoneTrigger”** para delimitar zonas especiales en las que sólo aquellos edificios que compartan tipo o los del tipo “wildcard” van a poder ser creados dentro del *trigger*.

Cualquier otra circunstancia que el diseñador tenga prevista para determinar si se ha de instanciar o no un edificio no está recogida en la plantilla y, por lo tanto, ya es responsabilidad del diseñador el implementarla.

5.21. Cambiar el nombre a la carpeta base

En ocasiones, al cambiar el nombre a la carpeta “StudentName” que encontramos directamente en la raíz “/Content” y sustituirlo por el *login* del diseñador, el motor a veces se confunde y muestra un error del estilo:



Lo que puede suceder, en primer lugar, es que al cambiar el nombre a la carpeta en la que están incluidos todos los assets de la *template* se desvinculen del “Project Settings” datos generales del proyecto como:

- El “GameModeBase” que, por defecto, era la clase “BP_GameModeBase”.
- El “GameInstance” que, por defecto, era la clase “BP_GameInstance”.
- Mapas por defecto, que tanto en “Editor Startup Map” como en “Game Default Map” era “TestMap_P”.

Por ello, **para corregirlo**, hay que seguir estos pasos:

1. Cambiar el nombre de la carpeta “StudentName” que contiene todos los assets.
2. Tratar de corregir todos los errores provocados con el cambio de nombre visualizando los redirectores en el propio “Content Browser”, “Filters”, “Other Filters”, **Show Redirectors** y corregirlos pinchando con el botón derecho del ratón sobre ellos y seleccionando la opción “Fix up”.

Este último era el paso clave, si tras esto último sigue sin funcionar:

1. Se cierra el editor.
3. Eliminar las carpetas “Intermediate” y “Saved”.
4. Regenerar el proyecto haciendo doble click en el archivo de extensión “.uproject”.
5. Ir a “Edit”, “Project Settings” y re asignar las clases: “BP_GameModeBase” (o el modo de juego que sea por defecto), “BP_GameInstance” y los mapas de juego por defecto.
6. En la pestaña principal, barra de herramientas localizada arriba del “Viewport”, “Blueprints”, “World Override” y seleccionar nuestro “BP_GameModeBase” o el “GameMode” específico por nivel”.

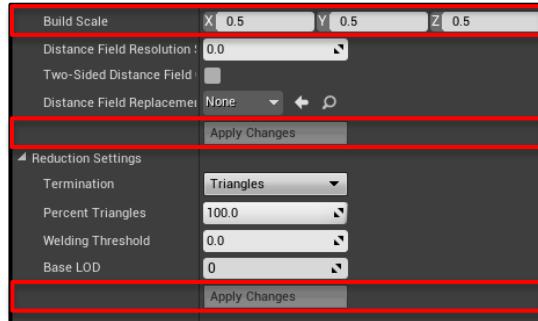
Una vez realizados estos pasos, todo debería de funcionar de nuevo puesto que este se trata de un “no error” que inexplicablemente el motor padece porque, digamos, se confunde al cambiar el nombre a esta carpeta core.

5.22. Escalado de una geometría y zoom dentro de ella

En el caso de que el diseñador importe un conjunto de assets en el proyecto para vestirlo visualmente, puede que algunos de las geometrías de los edificios sean excesivamente grandes o, por el contrario, resulten pequeñas. El desarrollador ha de recordar que, **en el editor de geometrías**, tiene la **opción “Build Scale”** para darle un



tamaño acorde a las métricas de su juego y, cuando lo haga, ha de darle al **botón “Apply Changes”**.

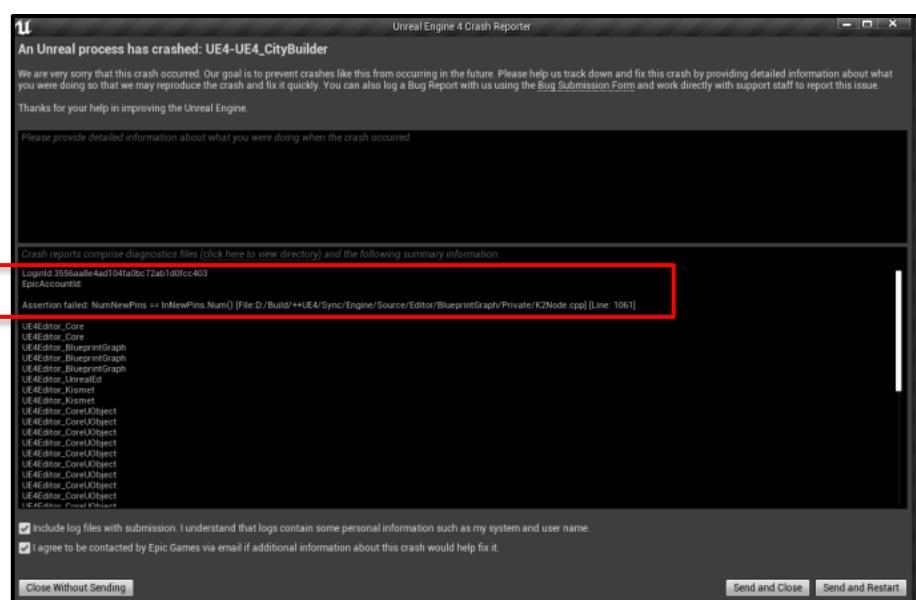


Por otra parte, en el caso de que la cámara llegue a poder meterse dentro de un edificio cuando el usuario realiza *zoom*, y tal y como se ha detallado en partes anteriores, hay que ajustar los **valores del epígrafe “Zoom” de la clase “BP_Pawn”** relativos al mínimo y máximo que puede alcanzar el “SpringArmComponent” –teniendo en cuenta, por supuesto, la altura a la que se coloca el objeto de la clase “BP_Pawn” sobre el mundo–.

5.23. Modificación de las estructuras de datos

En la plantilla existen algunas estructuras de datos pendientes de ser modificadas: “S_UniqueBuilding”, “S_ElementCost” y “S_TechCollection”. Estas estructuras, en el momento de entregar la plantilla, contienen únicamente datos a modo de ejemplo por lo que son muy altamente susceptibles de ser modificadas por el diseñador.

A la hora de que el desarrollador modifique la plantilla, por cuestiones internas del motor, **se desrecomienda totalmente que cambie el nombre a los datos ya existentes** en estas estructuras porque el motor sufrirá interrupciones en tiempo de modificación, mostrando visualmente el “Crash Report Client” un problema del estilo:



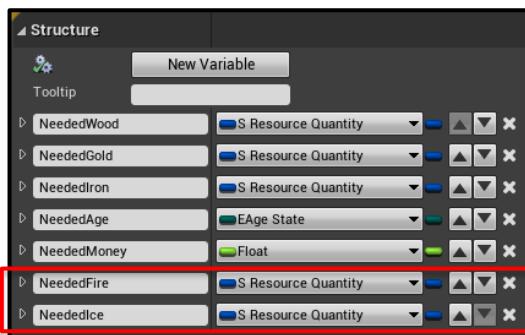
Es un error del motor que podemos bordear si, en lugar de modificar el nombre de un dato de una estructura de datos, el diseñador **añade los datos que necesita y elimina aquellos datos por defecto que le sobran**. El siguiente es un ejemplo real con la



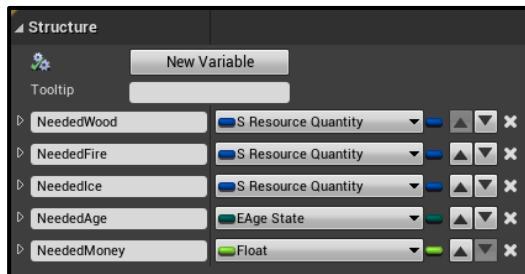
estructura “S_ElementCost”, utilizada para definir el coste en recursos de algunos elementos como edificios o tecnologías:



1: Datos entregados con la plantilla.

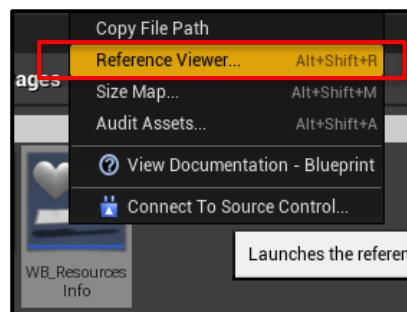


2: Se añaden nuevos datos.



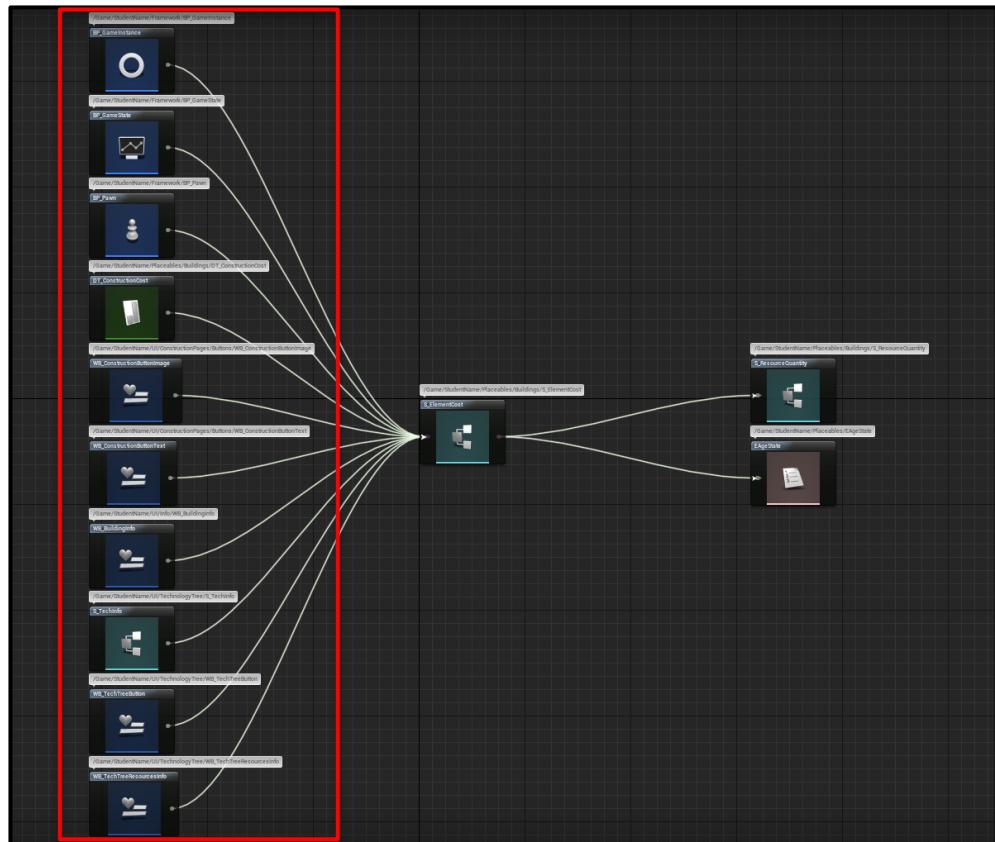
3: Se eliminan los datos de la plantilla que no se van a utilizar.

Es importante recordar que la **opción “Reference Viewer”** que encontramos sobre todos los objetos del “Content Browser” al pinchar sobre ellos con el botón derecho del ratón...

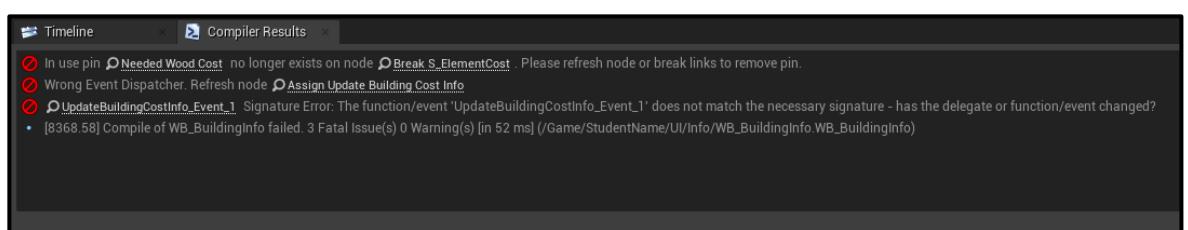




...permite al desarrollador revisar la lista de elementos que contienen variables del tipo de nuestra estructura de datos para, en definitiva, ver el listado de referencias:



Hay que **revisar uno a uno** pero, de toda esta lista, la mayoría sólo necesitan ser guardados de nuevo: allá donde pida recompilar es donde saldrá la lista de errores como por ejemplo el widget "WB_BuildingInfo" en el caso de la estructura "S_ElementCost":



5.24. Consideraciones finales

Esta guía ha tratado de dar solución de forma óptima a los principales problemas que se pueda encontrar el diseñador a la hora de adaptarse a ella para poder ser productivo y desarrollar su proyecto con un punto de partida ciertamente avanzado. En muchas ocasiones se trabaja con código o herramientas realizadas por terceras personas porque uno sólo no puede hacerlo todo y es importante entender del modo en el que el otro desarrollador ha llevado a cabo su tarea y, sobre todo, su idea.

Es por ello por lo que el objetivo de esta guía es ofrecer una buena panorámica general que permita una mejor adaptación a la plantilla. No obstante, es perfectamente posible



que, a lo largo de las distintas explicaciones y consideraciones recogidas a lo largo del presente documento, se haya obviado algún detalle por considerarlo menor, por irrelevante o, sencilla y llanamente, por olvido, es una pequeña posibilidad que existe.

Finalmente, se recomienda al diseñador, junto con la lectura y consulta de esta guía, que revise toda la funcionalidad entregada en la plantilla para ver de primera mano qué funcionalidad le es legada y qué ha de cambiar. Allá donde el diseñador vea en el código un “TODO” o un “TODO designer” es un punto susceptible de ser modificado de la forma que indica el comentario sobre el propio código o la documentación.