Grado en Diseño y Desarrollo de Videojuegos

Trabajo de final de grado

Septiembre 2020

Memoria

Tales from Vorkov City

Tutor:

Gustavo Patow

Departamento:

Informàtica, Matemàtica Aplicada i Estadístic

Area:

Llenguatges i Sistemes Informàtics (LSI)

Alumno:

Ángel Luis Moro Alonso

Dedicado a mi familia y amigos, los cuales siempre me apoyaron en todo desde el principio. Especialmente a mi madre, quien me animo a empezar esta etapa de mi vida, aunque ella no esté aquí, siempre estará conmigo en todo lo que haga.

Agradecimientos

Me gustaría agradecerle a Gustavo por ayudarme siempre tanto en el ámbito académico como fuera de él. Desde el primer momento, incluso antes de ser alumno de la universidad, siempre estuvo dispuesto a resolver todas mis dudas y ayudarme. Tengo claro que sin sus ánimos y su perseverancia conmigo no habría logrado llegar hasta aquí.

Índice

1. Introducción **20**

1.1. Descripción breve del problema 20

1.2. Objetivos 21

1.3. Tipología del proyecto 22

1.4. Alcances 22

1.5. Estructura del documento 23

2. Estudio de viabilidad **25**

2.1. Estudio del mercado 25

2.1.1. Plataformas de juego 25

2.1.2. Dispositivos móviles 27

2.1.2.1. Segmentación por plataforma 28

2.1.2.2. Segmentación por genero 30

2.1.2.3. Segmentación por idioma 32

2.2. Publico objetivo 33

2.2.1. Perfil del jugador 33

3. Planificación **35**

3.1. Planificación inicial 35

3.2. Planificación final 38

4. Marco de trabajo y conceptos previos **40**

4.1. Procedural Content Generation (PCG) 40

4.1.1. Metodos de generacion de contenido 40

4.1.2. PGC en el mundo de los videojuegos 41

4.1.3. PGC en la generacion puzzles 42

4.2. Conceptos previos 43

4.3. Referencias 43

4.3.1. Referencias para las mecánicas propias del juego 44

5. Diseño del videojuego **45**

5.1. Análisis de requisitos 45

5.1.1. Requisitos narrativos 45

5.1.2. Requisitos tecnológicos 45

5.1.3. Motor de juego 46

5.2. Género del videojuego 46

5.3. Mecánicas del juego 47

5.3.1. Toma de decisiones 47

5.3.2. Resolución de Puzzles 47

5.4. Narrativa 48

5.4.1. Sinopsis 48

5.4.2. Estructura narrativa 49

5.5. Estética y ambientación 50

6. Implementación y pruebas **52**

6.1. Laberinto 52

6.1.1. Explicación del algoritmo 52

6.1.2. Populate boarder 53

6.1.3. Spawn instances 57

6.1.4. Set initial tile 58

6.1.5. Bridges, pick nearby tile y backtrack 59

6.1.6. Populate walls and loop paths 64

6.2. Sistema de misiones 68

6.2.1. Estructuras de datos 69

6.2.2. Creación de misiones 71

6.3. Arboles de diálogo 76

6.3.1. Behaviour Tree 76

6.3.2. Blackboard 77

6.3.3. Task Nodes implementados 77

6.3.4. Decorators implementados 85

6.3.5. Behaviour Trees construidos 89

6.4. Inventario 95

6.4.1. Preparación 96

6.4.2. Añadir objetos 97

6.4.3. Existencia de un objeto en el inventario 99

6.4.4. Eliminar un objeto del inventario 100

6.4.5. Mostrar u ocultar el inventario 100

6.5. Movimiento y menú de interacción 102

6.6. Objetos interactuables 104

6.6.1. Recoger un objeto 105

6.6.2. Inspeccionar un objeto 105

6.6.3. Evento tick 106

6.6.4. Finalizar el diálogo 106

6.6.5. Hablar con un objeto 107

7. Resultados **108**

7.1. En función de los objetivos 108

7.2. Visión legal 112

8. Conclusiones **113**

9. Trabajo futuro **114**

Índice de figuras

1. Tabla de autoevaluación. 22
2. Distribución global del mercado de videojuegos 26
3. Gasto en videojuegos por dispositivo 27
4. Popularidad de los videojuegos en el mercado de aplicaciones 28
5. Representación de los jugadores de Android en el mercado global de móviles. 29
6. Clasificación de géneros de videojuegos según su penetración en el mercado. . 30
7. Clasificación de géneros de videojuegos según la fidelizacion de jugadores. . . 31
8. Crecimiento del mercado global de videojuegos. 32
9. Tipología de jugador [1] 33
10. Planificación inicial 37
11. Planificación final 39
12. Escenario de No Man's Sky. 42
13. El misterioso viaje de Layton. 43
14. Portada Mass Effect. 44
15. Árbol para representar la mecánica de decisiones. 47

16. Vorkov city. 50

17. Ejemplo para calcular posibles casillas. 53

18. Inicio de la construcción. 54

19. Función para rellenar las casillas. En las figuras siguientes la explicaremos paso a paso. 54

20. Casillas Norte. 55

21. Casillas Este. 55

22. Casillas Sur. 56

23. Casillas Oeste. 56

24. Creación de los modelos para muros exteriores. 57

25. Creación de instancias de los modelos. 57

26. Conversión de entero a casilla. 58

27. Definición de la casilla inicial. 59

28. Función para designar la casilla inicial. 59

29. Bridges, pick nearby tile y bactrack. 60

30. Visión global de Pick nearby tile. En las figuras siguientes la explicaremos paso a paso. 60

31. Inicio de Pick nearby tile. 61

32. Check north tile. 61

33. Cycle counter. 62

34. Add bridges & path tiles. 62

35. Check east tile. 62

36. Funcion Check east tile. 63

37. Final de Pick nearby tile. 63

38. Función Backtrack. 64

39. Función Convert Last Bridge. 64

40. Populate walls and loop paths. 65

41. Función Populate walls. 65

42. Función Add loops to path. En las Figuras siguientes las explicaremos paso a paso. 66

43. Inicio de la función Add loops to path. 66

44. Fin de la función Add loops to path. 67

45. Función Tile surronded. 67

46. Creación de los caminos. 68

47. Creación de muros con altura aleatoria. 68

48. Estructura de datos para los NPC. 69

49. Estructura de datos para los objetos. 70

50. Estructura de datos para las misiones. 71

51. Generar un bloque de misiones. 71

52. Obtener objetos e iterar sobre ellos. 72

53. Obtener datos del archivo,iterar y, si coinciden, los nombres añadir los datos. 72

54. Generar una misión. En las Figuras siguientes lo explicaremos paso a paso. . 73

55. Obtener un NPC aleatorio. 74

56. Obtener un objeto aleatorio. 74

57. Asignar misión a un NPC. 75

58. Obtener un NPC aleatorio distinto del primero. 75

59. Variables en el Blackboard. 77

60. Inicio del nodo de inicio de diálogo. 78

61. Final del nodo de inicio de diálogo. 78

62. Asignación del texto. 79

63. Controlador para cambiar de linea de texto. 79

64. Mostrar el mensaje de espera según el objeto. 80

65. Mostrar presentación. 80

66. Mostrar el mensaje de espera según el objeto. 81

67. Creando la Response List. 81

68. Mostrar respuestas. 82

69. Evento de respuesta. 82

70. Diferencia con el nodo T_AskQuestion. 83

71. Marcar misión activa. 83

72. Marcar primera misión activa. 83

73. Marcar misión de diálogo como activa. 84

74. Activando misiones de diálogo en ambos NPC. 84

75. Finalizando el diálogo. 85

76. Finalizando lógica del NPC. 85

77. Comprobación de una misión activa. 86

78. Comprobación entre la respuesta dada y la necesaria para entrar en esa rama. 86

79. Comprobación si un objeto está en el inventario. 87

80. Eliminación de un objeto del inventario. 87

81. Búsqueda de un objeto en el inventario. 88

82. Eliminación de un objeto en el inventario. 88

83. Comprobación si un NPC está a la espera de un objeto. 89

84. Visión general del árbol de diálogo para una misión de recogida de objetos. En las Figuras posteriores lo explicaremos paso a paso. 90

85. Inicio del Behaviour tree 91

86. Primera rama. 91

87. Segunda rama. 92

88. Tercera rama. 93

89. Visión general del árbol para iniciar una misión de diálogo. 94

90. Visión general del árbol para finalizar una misión de diálogo. 95

91. Atributos de un “slot“. 96

92. Inicio de un componente de inventario. 96

93. Preparación de un componente de inventario. 97

94. Añadir objeto al inventario. 97

95. Búsqueda de una pila de objetos en el inventario. 98

96. Crear una pila de objetos. 98

97. Añadiendo a una pila de objetos. 99

98. Existe el objeto en el inventario. 99

99. Eliminar un objeto del inventario. 100

100. Eliminar un objeto del inventario. 100

101. Comprobar si existe la ventana de inventario. 101

102. Creando la ventana de inventario. 101

103. Posicionamiento de la ventana de inventario. 102

104. Movimiento del personaje. 103

105. Limitación del movimiento del personaje. 103

106. Activación del modo menú. 104

107. Desactivación del modo menú. 104

108. Método para recoger un objeto. 105

109. Inspeccionar objeto. 106

110. Evento tick. 106

111. Exit dialogue. 107

112. Hablar con un objeto. 107

113. Primer laberinto generado. 108

114. Segundo laberinto generado. 109

115. Inicio de una misión de dialogo. 110

116. Final de una misión de dialogo. 110

117. Inicio de una misión de recogida de objetos. 111

118. Encontrado el objeto de la misión. 111

119. Finalizando la misión. 112

1. Introducción

Tales from Vorkov City es una aventura gráfica creada para dispositivos móviles. A lo largo del siguiente documento se detalla su diseño y su implementación.

Esta aventura se desarrolla en el año 2177 en Vorkov City, donde los robots están integrados dentro de la sociedad como una herramienta más para ayudar a los humanos en cualquier trabajo y necesidad que puedan tener. Entorno a la sede central de Vorkov se ha creado una megalópolis donde conviven humanos y aliens en total armonía. Vorkov City es un símbolo de innovación, progreso y estabilidad. Nuestro protagonista es James Jones, un afamado detective de la policía de Vorkov City que será el encargado de resolver el caso y así poder salvar la ciudad.

1.1. Descripción breve del problema

El genero de las **aventuras gráficas** normalmente presenta el gran problema de la rejugabilidad. Por regla general este tipo de juegos tan solo se juegan una vez ya que siempre son iguales.

Este problema genera que, para que un título de este genero ofrezca un numero de horas que satisfaga al jugador, se deben invertir muchos recursos.

Para resolver este problema los diseñadores recurren a varias técnicas. Por ejemplo:

- Introducir **objetos coleccionables**, lo que hace que, si el jugador no los obtuvo todos en su primera partida, lo intente de nuevo para completarlos todos.
- Una **historia ramificada** donde las decisiones del jugador forman la historia. Esta técnica es las más utilizada ya que el jugador tendrá curiosidad por saber qué hubiera

pasado si hubiera tomado otras decisiones, por lo que juega de nuevo tomando distintas decisiones.

Los desarrolladores y diseñadores generalmente no toman en cuenta uno de los aspectos claves de este género, que son los puzzles. El diseño de puzzles es costoso y por lo general en estas aventuras no cambian independientemente de las veces que se complete el título.

Por tanto los jugadores pueden disfrutar de una historia ramificada pero normalmente siempre tendrán que resolver los mismos puzzles, lo cual hace que la motivación de volver a completar la historia sea menor ya que no plantea ningún reto a superar.

1.2. Objetivos

El propósito de este proyecto es crear una aventura gráfica con puzzles generados proceduralmente para poder añadir **rejugabilidad**, junto con una historia ramificada, lo cual hará que el jugador pueda elaborar su propia historia sin repetir los mismos puzzles que ya hizo la primera vez.

Pretendemos así estimular al jugador a embarcarse en la aventura más de una vez sabiendo que los retos que encontrará no serán los mismo que las veces anteriores. Con esto conseguimos una mayor motivación a disfrutar de más horas de juego suponiendo un menor coste para los desarrolladores.

Este proyecto desarrollara un videojuego centrándose en la rejugabilidad y tratando de abaratar al máximo los costes del diseño y desarrollo.

1.3. Tipología del proyecto

Para poder cumplir los objetivos propuestos, los hemos dividido en varios apartados donde otorgaremos un porcentaje de la importancia de cada uno de ellos en nuestro proyecto.

- **Estética**: La estética elegida es futurista, para lo cual hemos adquirido unos recursos en Internet olvidándonos así del diseño.
- **Narrativa**: Necesitaremos escribir una narrativa donde existan múltiples opciones a lo largo de ella, pudiendo así adaptarse a las distintas elecciones que tome el jugador.
- **Mecánicas**: Diseñaremos e implementaremos las distintas mecánicas para que el jugador pueda completar los objetivos propuestos por la narrativa del juego. El jugador ha de ser capaz de resolver los distintos puzzles gracias a las mecánicas básicas de nuestro videojuego.
- **Tecnología**: El juego esta desarrollado para PC pero en un futuro prevemos su salida para el mercado Android.

Apartado	Peso
Estetica	0
Narrativa	0
Mecanicas	10
Tecnologia	90

Figura 1: Tabla de autoevaluación.

1.4. Alcances

El alcance general del proyecto consiste en generar una versión para demostración donde se muestren las mecánicas principales del juego (point & click, conversaciones y toma de

decisiones, historia ramificada) así como algunos tipos de puzzles generados proceduralmente.

1.5. Estructura del documento

La memoria está estructurada en las siguientes partes:

- **Capítulo 1**. Introducción: Se expone brevemente el problema, los objetivos, la tipología del proyecto y el alcance del mismo.
- **Capítulo 2**. Estudio de viabilidad: Justificamos la viabilidad del proyecto realizando un estudio de mercado y seleccionando el público objetivo.
- **Capítulo 3**. Planificación: Explicamos el desarrollo del proyecto a lo largo del tiempo, es decir, cuando llevaremos a cabo las distintas tareas que lo componen.
- **Capítulo 4**. Marco de trabajo y conceptos previos: Introducimos los conceptos técnicos previos para poder desarrollar el proyecto.
- **Capítulo 5**. Diseño del videojuego: Analizamos los requisitos necesarios para elaborar nuestro proyecto, seleccionamos las herramientas que utilizaremos y además exponemos distintos aspectos del mismo.
- **Capítulo 6**. Implementación y pruebas: Explicamos somo hemos implementado el proyecto.
- **Capítulo 7**. Resultados: Mostramos los resultados obtenidos.
- **Capítulo 8**. Conclusiones: Exponemos las conclusiones sobre los requisitos que hemos alcanzado, así como una evaluación general del proyecto.
- **Capítulo 9**. Trabajo futuro: Detallamos posibles mejoras o ampliaciones que podremos realizar en un futuro.

- **Capítulo 10**. Bibliografía: Listado de las referencias utilizadas en el proyecto.

2. Estudio de viabilidad

2.1. Estudio del mercado

Presentaremos un estudio de mercado donde analizaremos el estado actual del mercado de los videojuegos en las distintas plataformas para que podamos así comprender mejor las necesidades actuales y poder crear un producto que tenga éxito.

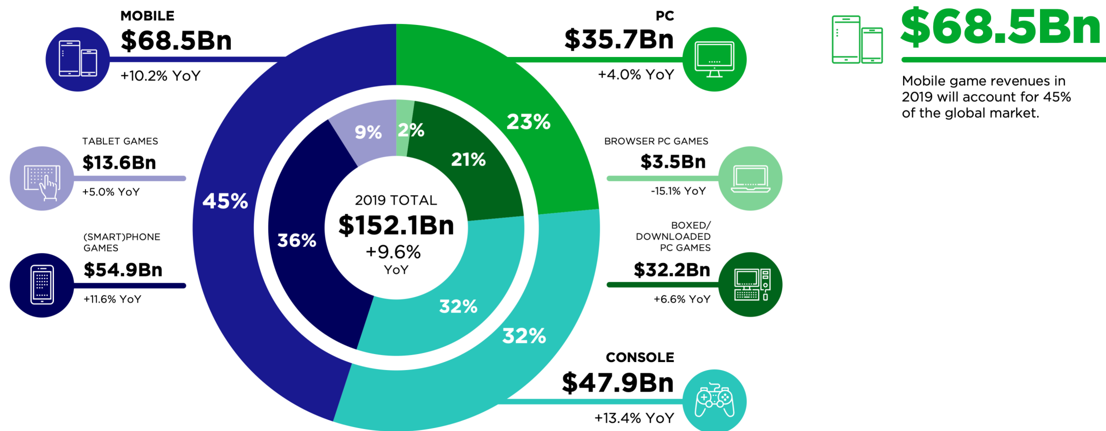
2.1.1. Plataformas de juego

A continuación analizaremos los beneficios en el mercado de videojuegos según el dispositivo en el que se juega, junto con su crecimiento anual.

Dentro del mercado de los videojuegos consideramos 3 grandes segmentos altamente diferenciados que son los videojuegos para PC, los que son para consola y los que son para dispositivos móviles.

2019 GLOBAL GAMES MARKET

PER DEVICE & SEGMENT WITH YEAR-ON-YEAR GROWTH RATES



Source: ©Newzoo | 2019 Global Games Market Report
newzoo.com/globalgamesreport

Figura 2: Distribución global del mercado de videojuegos

En 2019 el mercado de los videojuegos ha obtenido un total de 152 billones de dolares lo cual supone un incremento del casi 10 % respecto del año anterior. Ver figura 2.

Adentrándonos mas en los segmentos podemos observar que todos los segmentos y subsegmentos han crecido respecto al año anterior, salvo los juegos para navegador en el segmento de los videojuegos para PC, lo que nos hace intuir que el mercado de los videojuegos seguirá creciendo en años posteriores.

Desgranando por segmentos, el líder indiscutible es el mercado para dispositivos móviles con un 45 % de total del mercado destacando con un 36 % los smartphones dentro de él.

2.1.2. Dispositivos móviles

El mercado para dispositivos móviles es el que tiene mayor segmento del mercado en los videojuegos y ademas uno de los que mas rápidamente crece, como podemos ver en la Figura 3 [2].

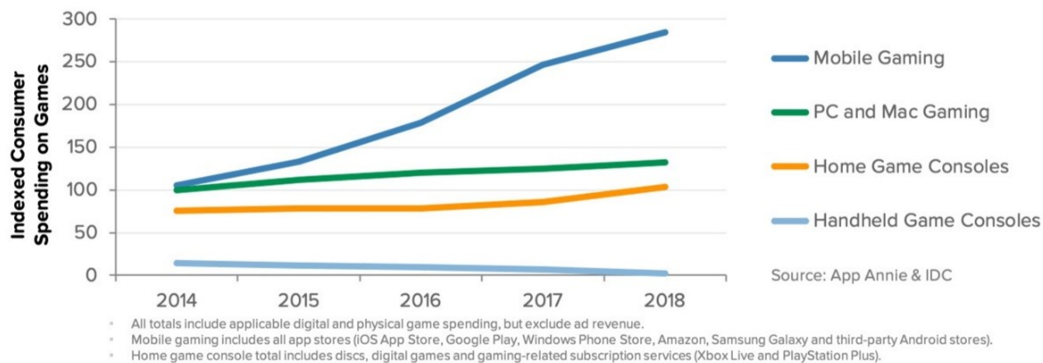


Figura 3: Gasto en videojuegos por dispositivo

El dinero que los consumidores se gastan en el mercado móvil se ha casi triplicado en 4 años, dejando al resto de mercados muy por debajo con un crecimiento mucho menor.

Para respaldar estos datos podemos hablar del videojuego Fornite el cual lanzo su juego para el mercado móvil en 2018 y el cual consiguió recaudar 3 millones de dolares en tan solo un día [3] y 100 millones de dolares en sus primeros 90 días IOS [4].

En base a estos datos, muchas empresas de videojuegos denominadas “indie”, debido a su bajo presupuesto, optan por desarrollar para este mercado en el que los costes son más bajos que en las otras plataformas.

2.1.2.1. Segmentación por plataforma

Dentro del mercado de telefonía móvil podemos distinguir dos grandes plataformas: Android e IOS. Dentro de ambas plataformas están presentes las aplicaciones de uso general y también los juegos.

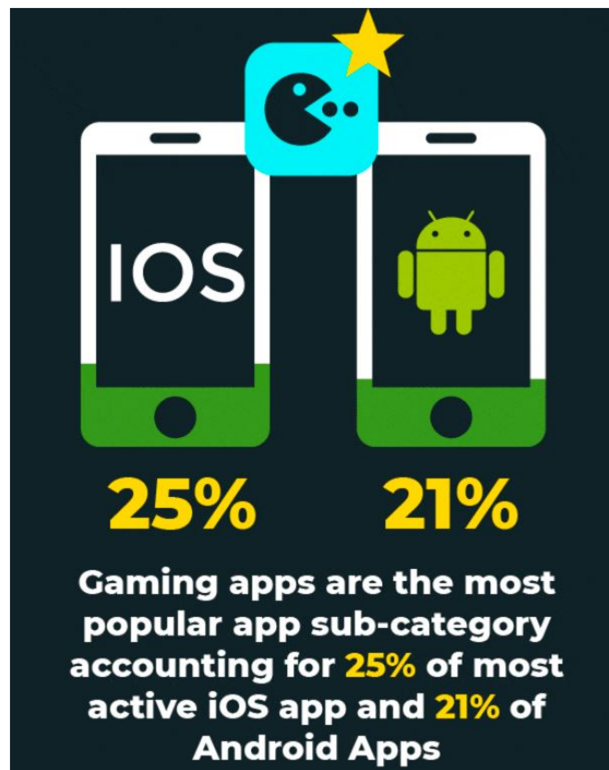


Figura 4: Popularidad de los videojuegos en el mercado de aplicaciones

Dentro del mercado de IOS los juegos ocupan un 25 % en cuanto a la subcategoría de aplicaciones más activas, mientras que en Android es un 21 %. Debemos valorar que son porcentajes dentro del mercado general, en números absolutos. La cantidad en Android es superior ya que su cuota de mercado es mayor.

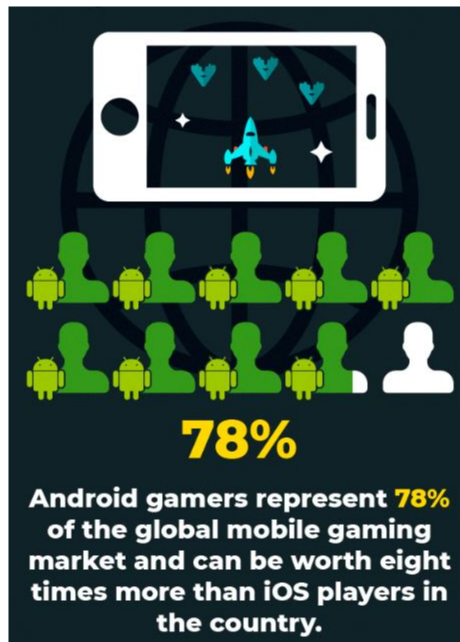


Figura 5: Representación de los jugadores de Android en el mercado global de móviles.

Dentro del mercado de jugadores, los de Android representan el 78 %, lo que es una cifra más que aceptable para decidirse por esta plataforma.

Si hablamos de costes de desarrollo en ambas plataforma, la plataforma iOS es la que más costes iniciales suele tener, ya que para un desarrollo óptimo precisa de un dispositivo MAC de sobremesa y un dispositivo iOS móvil. Ambos dispositivos tienen un precio elevado debido a que su único fabricante es Apple. En Android el coste de un terminal móvil oscila bastante pero con un terminal de gama media y un equipo de sobremesa se puede realizar el desarrollo con unos costes inferiores a los de iOS.Como dato también se aporta que es 8 veces mas rentable que el mercado en iOS que, si lo sumamos a que el desarrollo inicial para la plataforma Android es barato, el margen de beneficios aumenta considerablemente para un producto en Android.

2.1.2.2. Segmentación por genero

El producto que vamos a desarrollar es del género aventura y puzzles. Es un producto nuevo y para saber qué podemos esperar en su lanzamiento, analizaremos como estos géneros son aceptados dentro del mercado móvil por los usuarios.

Figura 6: Clasificación de géneros de videojuegos según su penetración en el mercado.

La Figura 6 nos muestra como el primero de todos los generos en cuanto a penetración en el mercado son los juegos de puzzles, mientras que los juegos de aventura están en la sexta posición.

Esto destaca la importancia de los puzzles dentro de nuestro proyecto, indicando que debemos potenciar la faceta de los puzzles para tener una mejor acogida dentro del mercado.

La acogida en el mercado es la primera fase de todo producto, pero una vez ha salido al mercado hay que estudiar cual es la fidelidad de los usuarios a los productos. Lo analizaremos por plataforma y por género.

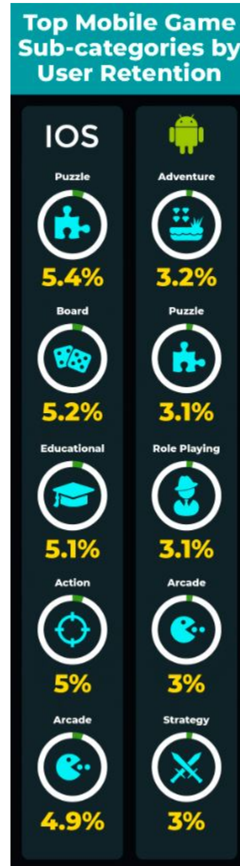


Figura 7: Clasificación de géneros de videojuegos según la fidelizacion de jugadores.

El género de puzzles es el número 1 en la plataforma iOS y el número 2 en la plataforma Android, siguiendo de cerca al género de la aventura.

Con estos datos podemos observar que los dos géneros de nuestro producto son los primeros en cuanto a la retención de usuarios en Android, por lo que podemos pronosticar que tras el lanzamiento conservaremos un gran número de usuarios.

2.1.2.3. Segmentación por idioma

En un videojuego es importante el idioma ya que dependiendo del idioma en el que esté el contenido, podemos aspirar a llegar a un distinto números de usuarios. El inglés es el lenguaje mas estándar dentro de la industria, pero no por este motivo es el que más jugadores hablen de forma nativa.

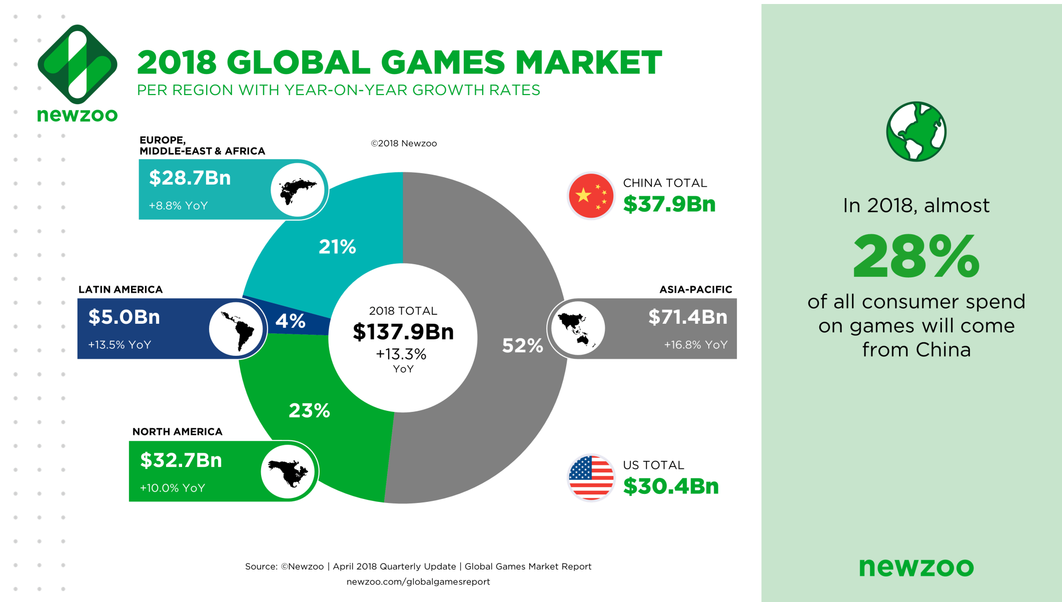


Figura 8: Crecimiento del mercado global de videojuegos.

Según los datos mostrados en la Figura 8, las regiones con mayor gasto en videojuegos son Asia y el pacífico seguidos por Norteamérica y Europa junto con África y el Medio-Este.

Dentro de estas regiones destaca el mercado chino y el mercado estadounidense, por lo que podemos deducir que el idioma chino y el inglés son los mas rentables según este estudio.

2.2. Publico objetivo

2.2.1. Perfil del jugador

Para identificar el perfil del jugador nos hemos basado en la clasificación que hace Richard Bartle. Bartle clasifica el perfil de los usuarios según la personalidad y los comportamientos que muestran en los juegos. Según Bartle, podemos encontrar cuatro perfiles distintos de usuario :

- **Achievers**: tienen como objetivo resolver retos con éxito y conseguir una recompensa por ello.
- **Explorers**: quieren descubrir y aprender cualquier cosa nueva o desconocida del sistema.
- **Socializers**: sienten atracción por los aspectos sociales por encima de la misma estrategia del juego.
- **Killers**: buscan competir con otros jugadores.

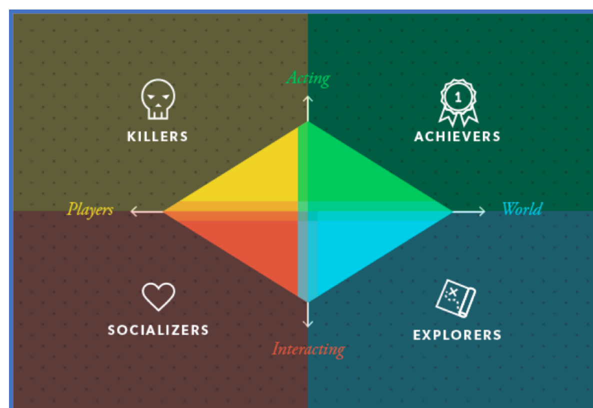


Figura 9: Tipología de jugador [1]

Basándonos en la figura 9, identificamos a nuestros jugadores con al tipologia de Achievers debido a que el principal objetivo del género es de resolver puzzles. Ademas también consideramos que los jugadores son en parte explorers ya que deben explorar el mundo que se encuentran para aprender a superar los retos planteados.

3. Planificación

3.1. Planificación inicial

La planificación del proyecto se divide principalmente en 3 etapas:

1. **Fase de diseño**: En esta fase planteamos todo el diseño de nuestro juego y tomamos las decisiones sobre el motor a utilizar, las mecánicas elegidas, estética, etc. Esta fase es una de las más importantes porque definiremos todo el proyecto asentando así como las bases para los siguientes pasos. Dentro de esta fase definimos las siguientes taréas:
    a) Definición de la idea
    b) Elaboración de los requisitos
    c) Definición del alcance del proyecto
    d) Elaboración del GDD
2. **Fase de investigación**: Una vez establecidas las bases del proyecto e identificadas las técnicas que utilizaremos, procedemos a investigar acerca de ellas para aplicarlas en nuestro proyecto. Un buen trabajo de investigación en esta fase nos ayudará a una mejor implementación de las técnicas en nuestro proyecto.
    a) Recopilación de información
    b) Estudio de las técnicas en Unreal Engine 4
    c) Técnicas procedurales
3. **Fase de desarrollo**: Aquí culminaremos todo lo planeado en la primera fase y pondremos en práctica lo aprendido en la fase de investigación. Nos meteremos de lleno con Unreal Engine para implementar todo lo necesario para nuestro proyecto.

a) Mecánicas básicas

b) Generación procedural de laberinto

c) Generación procedural de misiones

d) Testeo

Ver Figura 10.

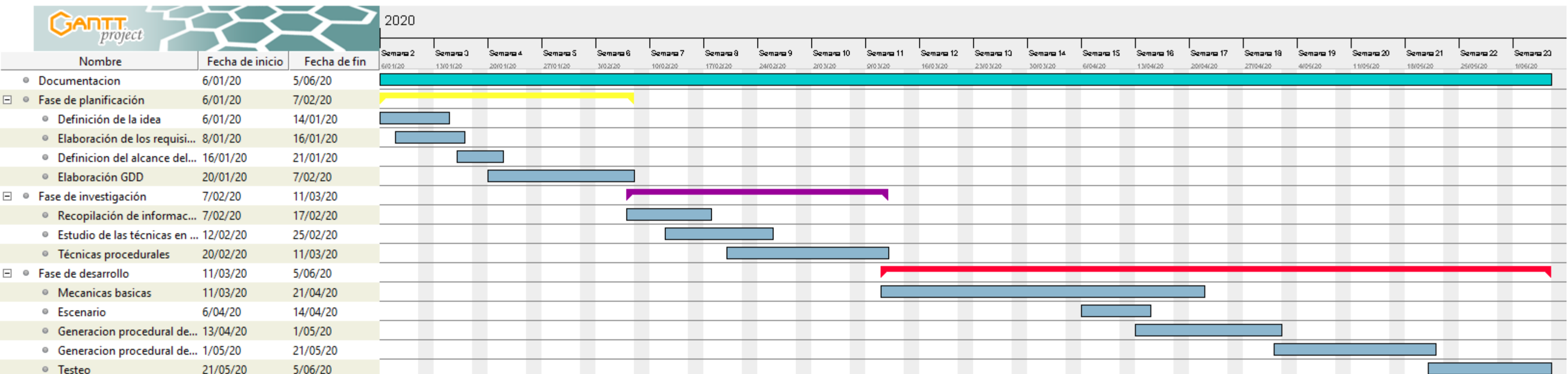


Figura 10: Planificación inicial

3.2. Planificación final

En cualquier proyecto puede surgir imprevistos que modifiquen la planificación inicial. En nuestro caso el principal motivo es la pandemia mundial asociada al COVID-19. Si a esto le sumamos que algunas tareas han requerido más tiempo del que estimamos en primera instancia, obtenemos como resultado una nueva planificación. Ver Figura 11.

Las tareas que mas tiempo me han llevado son:

1. Elaboración GDD
2. Técnicas procedurales
3. Generación procedural de laberinto
4. Generación procedural de misiones

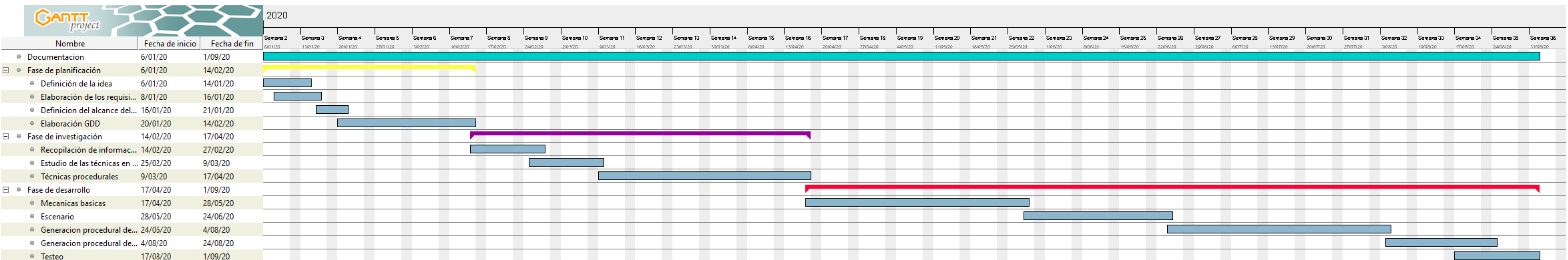


Figura 11: Planificación final

4. Marco de trabajo y conceptos previos

En esta sección hablaremos acerca de distintos conceptos que se han de conocer para después hacer un buen seguimiento del proyecto.

4.1. Procedural Content Generation (PCG)

Uno de los mayores costes del desarrollo de un videojuego es la creación de contenido. Algunos investigadores estiman que es alrededor del 30 % - 40 % del presupuesto de un juego AAA. Este factor hace que se necesiten personal altamente especializado, limitando asi las posibilidades de los pequeños estudios y repercutiendo en grandes costes para las grandes desarrolladoras.

La generación procedural de contenido es a lo que nos referimos cuando generamos datos de forma algorítmica y no manual, lo que trata de solucionar el gran problema de la creación de contenido en la industria del desarrollo de videojuegos. Esto tiene muchas aplicaciones en varios campos como en el de la Informática Gráfica, donde se puede usar para generar edificios, o en el campo de los videojuegos para la generación de escenarios, etc.

4.1.1. Metodos de generacion de contenido

En el mundo de la generación procedural de contenido podemos distinguir 3 grandes categorias en cuanto al método de generación se refiere.[2]

1. **Métodos tradicionales**: Están basados en la generación pseudoaleatoria de números, lo que los hace métodos rápidos y sencillos de diseñar. Es un método determinista, lo que significa que, dada una semilla, siempre obtendremos el mismo resultado para esa

misma semilla, por lo que sabremos el resultado si conocemos la semilla.

2. **Métodos basados en búsqueda**: Métodos basados en generar contenido para después evaluarlo. Generalmente, una vez el contenido ha sido creado, se le otorga una puntuación en base a ciertos criterios establecidos por el diseñador del algoritmo. Una vez generados, normalmente se selecciona el contenido con mejor puntuación o que cumplan un mínimo de requisitos [5].

3. **Machine learning**: Consiste en la generación de contenido basándose en un conjunto de datos previamente introducidos. Esta generación de contenido se produce a través de un modelo entrenado previamente con técnicas de Machine Learning. Algunos de estos algoritmos son redes neuronales, métodos probabilisticos, árboles de decisión ,etc. La generación puede ser parcial o completa, autónoma, interactiva o guiada. El contenido generado puede ser cualquiera dentro del juego, desde mapas a misiones u objetos [5].

4.1.2. PGC en el mundo de los videojuegos

Uno de los usos mas comunes de esta técnica en los videojuegos es para la **generación de escenarios** partiendo de una semilla básica con la cual el algoritmo genera todo lo necesario para construcción del escenario según las restricciones puestas por los creadores del mismo. Como ejemplos de esta aplicación en el mundo de los videojuegos tenemos títulos como Minecraft(2011) y No Man's Sky(2016) ambos pertenecientes al genero del sandbox y bastante conocidos por sus escenarios en el mundo del videojuego. En No Man's Sky la generación procedural de escenarios se lleva un paso mas allá de lo visto hasta el momento, consiguiendo generar universos enteros de esta forma [6]. Ver Figura 12.

Figura 12: Escenario de No Man's Sky.

4.1.3. PGC en la generacion puzzles

Centrándonos en el genero de puzzles, esta técnica no es tan usada como la comentada anteriormente. Esto es debido a que los puzzles presentan unas restricciones mucho más fuertes para ser generados que un escenario. Otro factor es que cada tipo puzzle posee unas reglas distintas para resolverlo, lo que implica que no existe un algoritmo único que genere todo tipo de puzzles. Esto convierte a la generación de puzzles de forma procedural en un gran área de investigación.

Dentro de la generación procedural en los puzzles podemos distinguir dos formas de hacerlo, la generación online y la generación offline. La generación online es aquella que se realiza mientras el usuario está jugando, aportando así una gran variedad de contenido, mientras que la generación offline es la que se produce mientras se está desarrollando el videojuego. Para que un algoritmo pueda ser usando para generar puzzles mientras el jugador está jugando, debe ser un algoritmo rápido y muy preciso para poder general los puzzles de

manera adecuada para la experiencia del jugador.

4.2. Conceptos previos

- **Point&Click**: son aquellos juegos donde se usa exclusivamente el ratón del PC para interactuar con el juego, siendo necesario hacer click en distintas zonas del escenario para realizar las acciones.

4.3. Referencias

Las referencias principales son las aventuras graficas, principalmente Grim Fandango, Monkey island y la saga del Profesor Layton (vr Figura 13).

Figura 13: El misterioso viaje de Layton.

Estos títulos son aventuras gráficas de tipo point and click, y comparten las mecánicas típicas de este género. Estas mecánicas son la exploración y la resolución de puzzles, con la

combinación de objetos, ya sea del inventario o del escenario.

4.3.1. Referencias para las mecánicas propias del juego

En cuanto a la mecánica de toma de decisiones y actitud del personaje principal, tomamos como referencia Mass Effect. Mass Effect es una saga de juegos de la compañía Bioware donde nos metemos en la piel de un protagonista que, según las decisiones que tome a lo largo de la historia, cambiará la actitud que tienen los personajes hacia el, las opciones de diálogo e incluso cambia su aspecto físico. Ver Figura 14.

Figura 14: Portada Mass Effect.

5. Diseño del videojuego

5.1. Análisis de requisitos

Para este proyecto consideramos dos grandes bloques en cuanto al análisis de los requisitos para su realización. El primer bloque son los requisitos narrativos, que son aquellos que afectan directamente al desarrollo de la historia, y el segundo el bloque tecnológico, que es aquel que nos indica las necesidades a nivel técnico del proyecto.

5.1.1. Requisitos narrativos

Al tratarse de una **historia ramificada** necesitamos diseñar las distintas variantes de la misma en función de las decisiones del jugador. Además, hemos de añadir a la ecuación el componente de aleatoriedad que aporta el sistema, para así poder tener una historia diferente cada vez que se juega. Los principales requisitos narrativos son:

- Historia con distintas lineas argumentales y distintos finales.
- Módulos narrativos intercambiables.
- Coherencia y cohesión entre la narrativa.
- Atractiva para el jugador aunque juegue más de una vez.

5.1.2. Requisitos tecnológicos

Nuestro juego esta pensado para lanzarse en dispositivos móviles y PC. En este momento la prioridad es lanzarlo para PC por lo que nos centraremos en los requisitos de esta plataforma.

- Generación procedural de puzzles.
- Mecánicas básicas para la resolución de los puzzles.
- Interfaz y mecánicas intuitiva para el jugador.

5.1.3. Motor de juego

El motor de juego elegido es **Unreal Engine 4**. Unreal es un motor de juego creado por Epic Games en 1998 para su shooter llamado Unreal. A día de hoy es uno de los grandes motores de la industria del videojuego, junto con Unity.

Dentro del mundo del desarrollo de videojuegos, Unreal suele utilizarse más para el desarrollo de videojuegos en 3D y para consolas o PC, por lo que se ajusta más a nuestro objetivo de lanzarlo en PC.

Ambos motores tienen documentación en sus correspondientes páginas web, pero en el caso de Unreal ademas añade tutoriales específicos, proyectos de ejemplo, así como una comunidad muy activa para resolver cualquier problema. Este punto es importante cuando se quieren desarrollar características no tan comunes en los videojuegos, como es nuestro caso.

5.2. Género del videojuego

El género del videojuego que planteamos es una **aventura gráfica** de tipo point and click, con el género secundario de puzzles.

Una aventura gráfica podría definirse como un tipo de videojuego en el que se avanza a lo largo de la historia resolviendo distintos puzzles que ayudan a avanzar la trama del jueg,o

junto con la interacción con objetos o personajes.

5.3. Mecánicas del juego

El juego tendrá dos mecánicas principales, la primera será la toma de decisiones que influirá en el transcurso de la historia y la segunda será la resolución de puzzles para poder avanzar en la trama del juego.

5.3.1. Toma de decisiones

Al jugador se le mostrarán una serie de opciones donde deberá elegir una, la cual repercutirá en eventos posteriores de la historia. Las decisiones, ademas de influir en la historia, modificarán la actitud del personaje hacia un carácter más intimidador, o más negociante, lo cual le permitirá elegir unas opciones distintas dentro del juego, así como condicionara sus relaciones con otros personajes. Ver Figura 16.

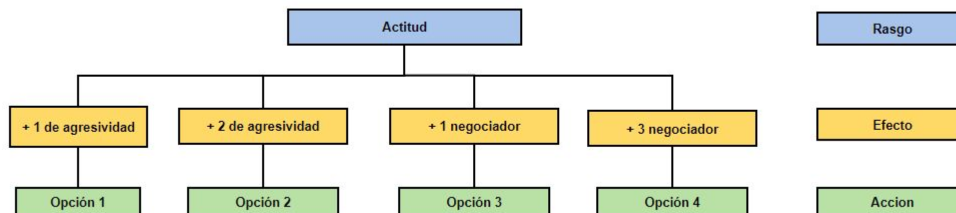


Figura 15: Árbol para representar la mecánica de decisiones.

5.3.2. Resolución de Puzzles

La resolución de puzzles será necesaria para el avance de la trama narrativa en el juego. El jugador deberá resolver los distintos tipos de puzzles en los cuales algunos le aportaran

pistas para resolver el caso, o simplemente serán necesarios para avanzar.

Dentro de los puzzles podemos distinguir dos tipos: los **puzzles narrativos** y el resto de puzzles. Los puzzles narrativos son aquellos que forman parte de la progresión de la narrativa. La solución implica la exploración y el pensamiento lógico, así como creativo [5].

El resto de puzzles estan formados por varios puzzles generados de forma procedural, y de varios tipos que se le plantearan al jugador en el transcurso de la historia. Los tipos de puzzles que se plantearan serán los siguientes:

- Narrativos
- Laberintos
- Sokoban

5.4. Narrativa

5.4.1. Sinopsis

Año 2152. El científico Dorian Eustace Vorkov consigue crear el primer robot totalmente autónomo y funcional para ayudarle en sus tareas domésticas. Este gran avance en el campo de la robótica revoluciona la comunidad científica.

Año 2163. Los robots son empleados en la exploración espacial obteniendo la respuesta a la pregunta "¿Estamos solos?". La humanidad descubre que no son la única raza inteligente. En los años posteriores la Tierra se convierte en un mundo habitado por todo tipo de especies.

25 años después la corporación Vorkov fabrica más de 1000 tipos distintos de robots para todo el mundo. Los robots están integrados dentro de la sociedad como herramientas para

ayudar a los humanos en cualquier trabajo y necesidad que puedan tener.

Jeremy Jones, JJ para sus amigos, es un detective de la policía de Vorkov City que se verá envuelto en una trama de intrigas y lucha de poderes para tratar de resolver el caso más grande de toda su carrera.

5.4.2. Estructura narrativa

La historia es ramificada por lo que la progresión de la misma la definirá el jugador dependiendo de las decisiones que tome cuando se le presenten varias opciones para elegir. La historia completa la dividiremos en episodios, donde las decisiones de los episodios anteriores tienen un peso en los siguientes. Cuando decimos de que las decisiones son importantes, nos referimos a que, si los jugadores deciden dejar morir a otro personaje, ese personaje no aparecerá más en la trama. También, si los jugadores muestran una actitud hostil hacia un personaje, es posible que éste no le ayude a conseguir sus objetivos o incluso le ponga trabas.

A continuación, mostramos la estructura narrativa dividida en episodios:

1. **Introducción**: En este primer episodio haremos que los jugadores se familiaricen con las mecánicas básicas del juego, así como intentar que empaticen con nuestro protagonista. En la mitad del episodio plantearemos el nudo de la trama y en el final dejaremos varias cuestiones sobre la trama sin contestar para mantener el interés de los jugadores.

2. **Desarrollo**: Este episodio es el más largo donde desarrollaremos la trama principal en mayor medida. Plantearemos bastantes decisiones y puntos de inflexión a los jugadores para así poder tener una mayor variedad de desenlaces posibles de la historia.

3. **Desenlace**: en este punto los jugadores ya han tomado bastantes decisiones por lo que basándonos en ellas iremos construyendo el desenlace de la historia.

5.5. Estética y ambientación

El juego está ambientado en el 2077 donde los robots y los alienígenas conviven en la Tierra con los humanos. Para este proyecto hemos elegido una ambientación **futurista** que de adecua al año en el que se desarrolla. El escenario es una ciudad del año 2077 donde se desarrollará toda la trama. Predominan los neones, los colores vivos y los grandes carteles publicitarios como principal exponente para generar una ambientación de un futuro más avanzado tecnológicamente

La ambientación futurista esta combinada con una estética "lowpoly". Cuando hablamos de "lowpoly" nos referimos a unos modelos 3D con pocos polígonos y, por tanto, con una apariencia poco realista. Hemos elegido esta estética porque, además de necesitar un menor nivel de recursos para funcionar (importante en dispositivos móviles), también nos aporta una estética más desenfadada, lo que nos permite incluir más humor en la trama.

Figura 16: Vorkov city.

Todos los modelos 3D y los personajes pertenecen a Sinty Studios, concretamente a su paquete “POLYGON - Sci-Fi City Pack“ [7] .

6. Implementación y pruebas

Nuestro proyecto está desarrollado en Unreal Engine 4 y todo el código está en formato blueprint (programación visual de Unreal). Las blueprint suelen ser bastante extensas, por lo que comentaremos poco a poco el flujo de cada una con imágenes parciales de la misma, acompañadas de explicaciones.

6.1. Laberinto

El laberinto que hemos implementado se genera de manera **procedural**. Gracias a esto podemos tener una infinidad de distintos laberintos sin tener que diseñarlos uno a uno. Para facilitar la explicación de cómo hemos implementado esta funcionalidad, la hemos dividido en fases.

6.1.1. Explicación del algoritmo

Para la implementación del laberinto procedural hemos decidido utilizar el algoritmo conocido como **backtracking**. Gracias a él vamos construyéndolo, y si llegamos a un punto donde es inviable seguir, podemos volver atrás y explorar otras soluciones. Concretamente, lo utilizamos para crear los posibles caminos dentro del laberinto.

Para encontrar los posibles caminos, tenemos que definir una manera de hacerlo. Nosotros hemos decidido usar los 4 puntos cardinales (Norte, Sur, Este y Oeste) y además usar 2 casillas por cada uno. Usar 2 casillas significa que entre nuestra casilla actual y la casilla de destino tendremos otra que llamaremos puente o "bridge"(ver Figura 17). Partiendo de esta base obtenemos 4 ecuaciones para calcular cual será la siguiente casilla en función de la dirección. Si consideramos c como la casilla actual en un plano donde x es la fila (comenzando

en 0 en la esquina superior izquierda), las ecuaciones son las siguientes:

1. Norte (arriba): $c - 2x$
2. Este (derecha): $c + 2$
3. Sur (abajo): $c + 2x$
4. Oeste (izquierda): $c - 2$

0	1	2	3	4	5	...
x	X+1	X+2	X+3	X+4	X+5	...
2x			C-2x			
3x						
4x	C - 2		C		C + 2	
5x						
...			C+2X			

Figura 17: Ejemplo para calcular posibles casillas.

6.1.2. Populate boarder

Para comenzar llenaremos una estructura llamada "board tiles" con números enteros. Esta estructura representa los bordes exteriores de nuestro laberinto. También podemos observar un condicional, que nos sirve para mostrar dichos bordes u ocultarlos a la vista del jugador. Ver Figura 18.

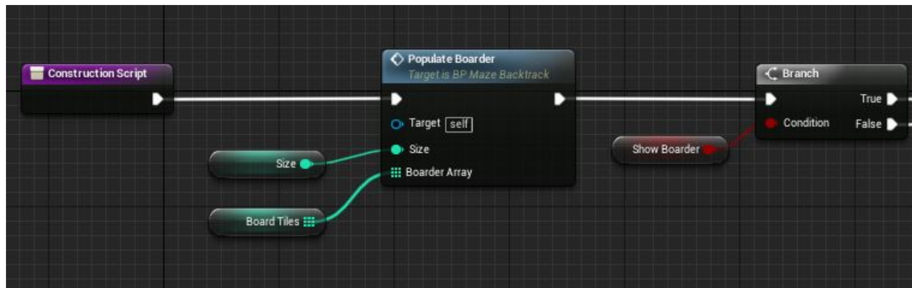


Figura 18: Inicio de la construcción.

En la Figura 18 podemos observar la función “populate boarder“. Esta función es la que rellena la estructura con las casillas que necesitamos.

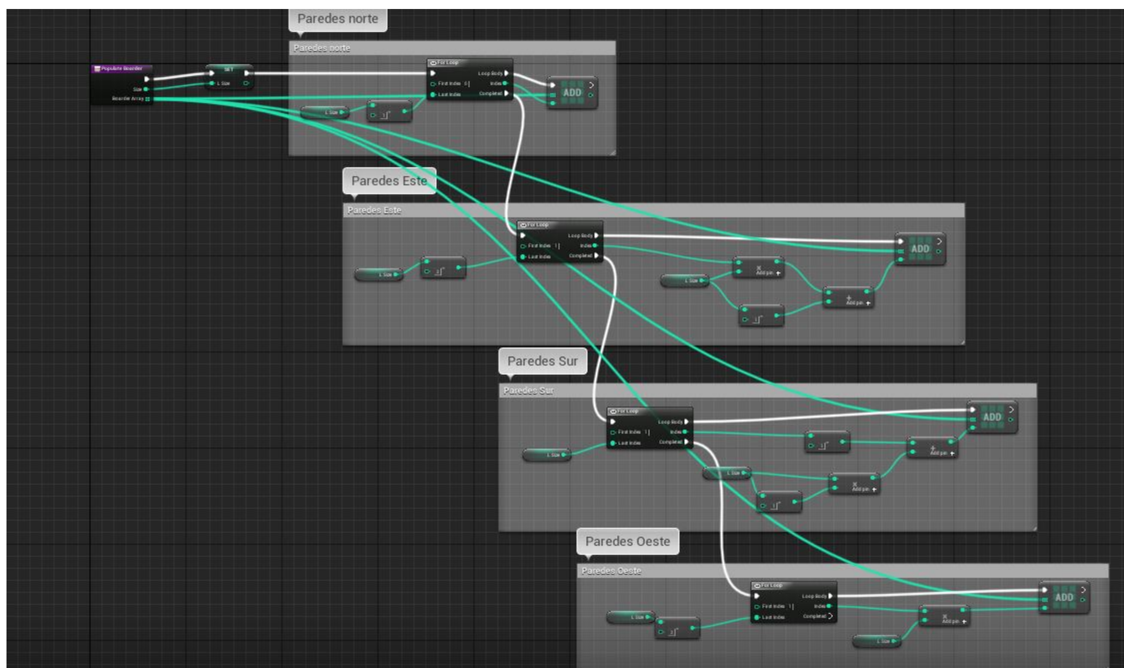


Figura 19: Función para rellenar las casillas. En las figuras siguientes la explicaremos paso a paso.

En la Figura 19 desarrollamos la función nombrada anteriormente. Ésta se compone de 4 bucles, donde cada uno de ellos se ocupa de una zona del laberinto. Comenzamos con las

casillas superiores (Norte) que son las casillas de la número 0 hasta el tamaño del laberinto menos uno. Ver Figura 20.

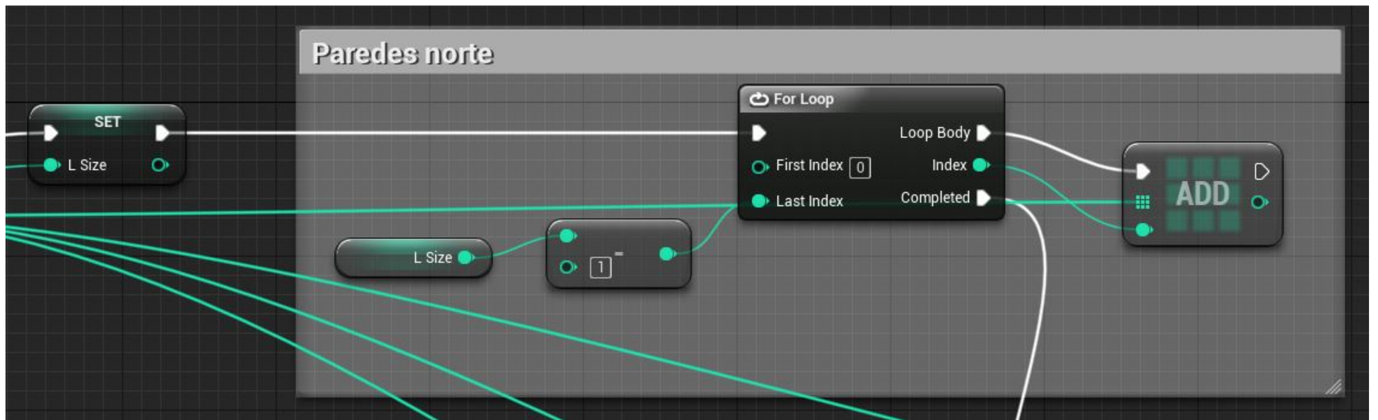


Figura 20: Casillas Norte.

A continuación, seleccionaremos las paredes de la derecha (Este). Para ello comenzamos en la segunda fila (la casilla de la primera fila pertenece a la parte Norte) e iteramos por el borde derecho. Ver Figura 21.

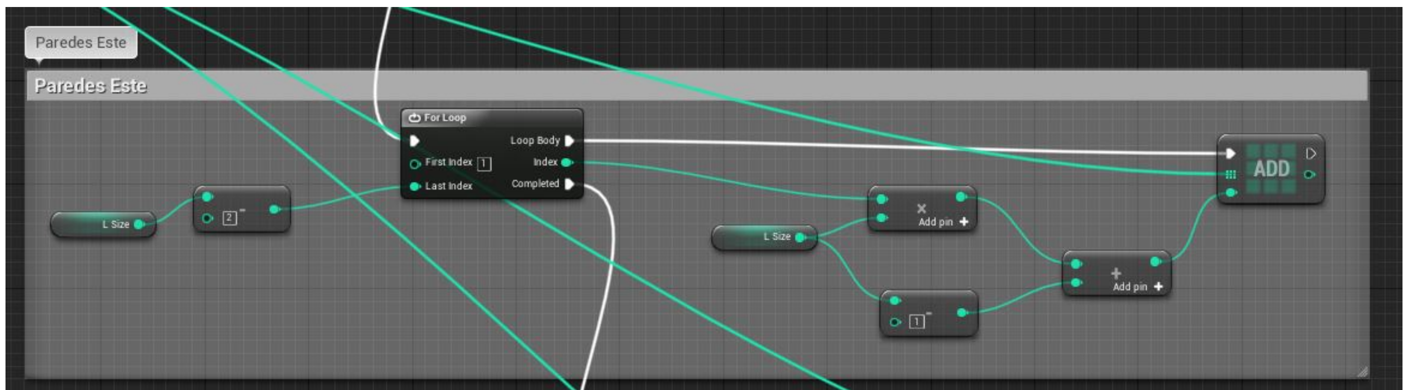


Figura 21: Casillas Este.

Para las paredes del sur comenzaremos a partir de la segunda casilla de la última fila (la

primera casilla pertenece a la zona Oeste) e iremos añadiendo las casillas hasta la última. Ver Figura 22.

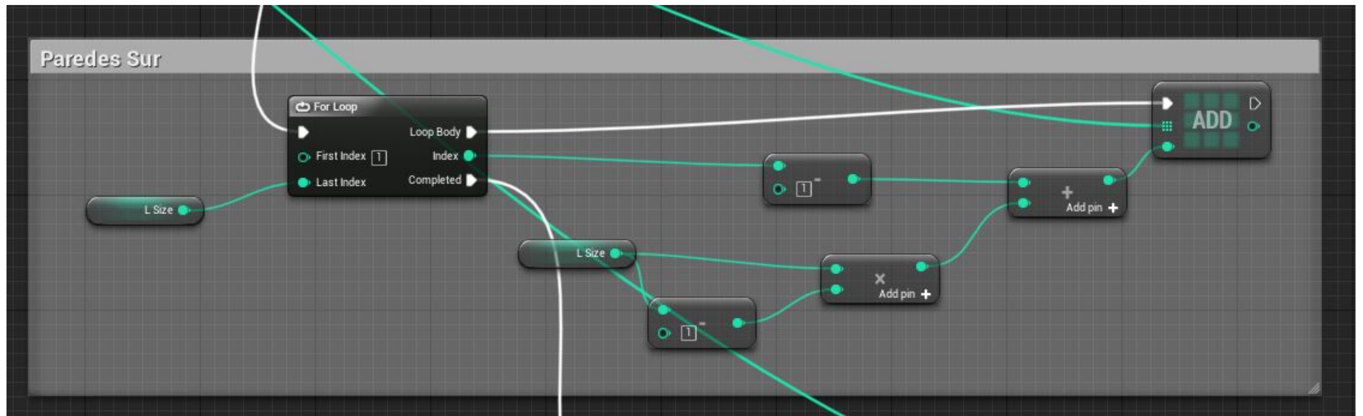


Figura 22: Casillas Sur.

Finalmente, las paredes de la izquierda (Oeste) las rellenamos recorriendo la columna desde la segunda fila (la casilla de la primera fila corresponde al Norte). Ver Figura 23.

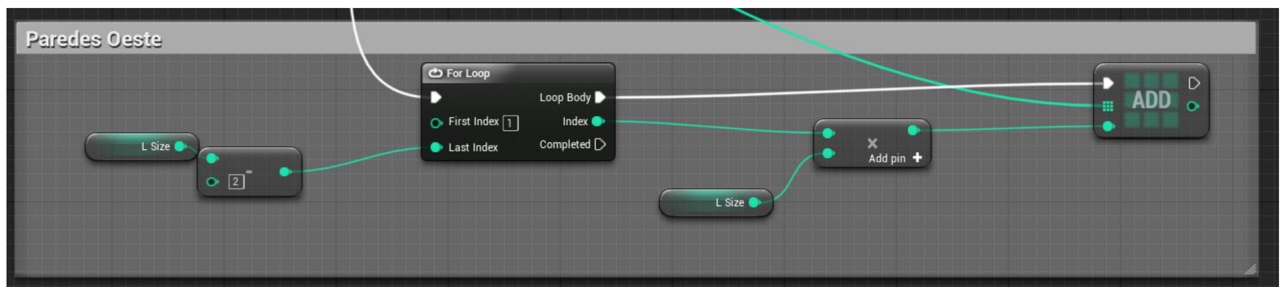


Figura 23: Casillas Oeste.

6.1.3. Spawn instances

Una vez tenemos definidas las paredes exteriores de nuestro laberinto, tenemos que crear los modelos 3D que corresponden a los muros exteriores. Para ello hemos creado la función “Spawn instances“ que se encarga cargar e instanciar los modelos 3D que le indiquemos, con el tamaño y la posición indicados.

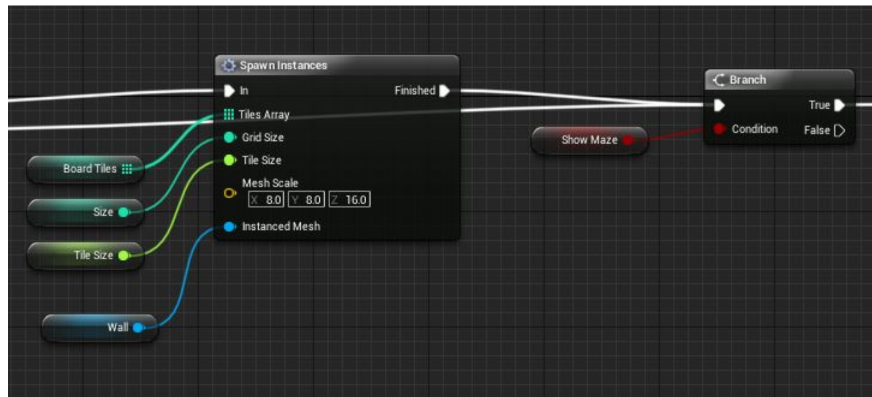


Figura 24: Creación de los modelos para muros exteriores.

En la Figura 24 podemos observar la función ”spawn instances”. Esta función crea una instancia del modelo elegido en base la estructura proporcionada. Para ello comenzamos recorriendo la estructura que tenemos con números enteros, después convertimos cada número a su correspondiente casilla (con coordenadas x e y) para, por último, crear el modelo. Ver Figura 25.

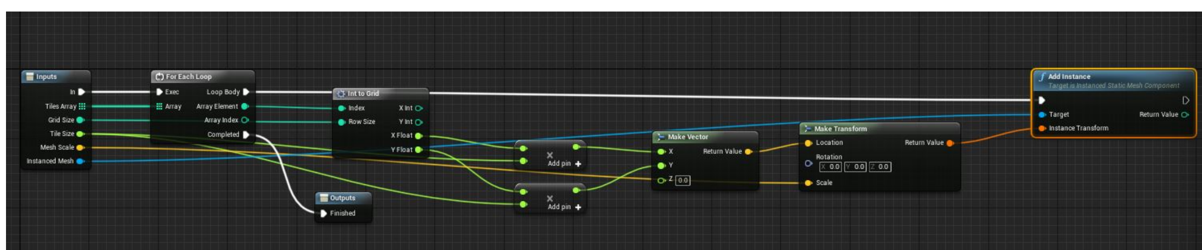


Figura 25: Creación de instancias de los modelos.

Para convertir el índice que nos indica en que casilla estamos, hemos elaborado una macro que convierte el índice a la posición X e Y correspondiente. Ver Figura 26.

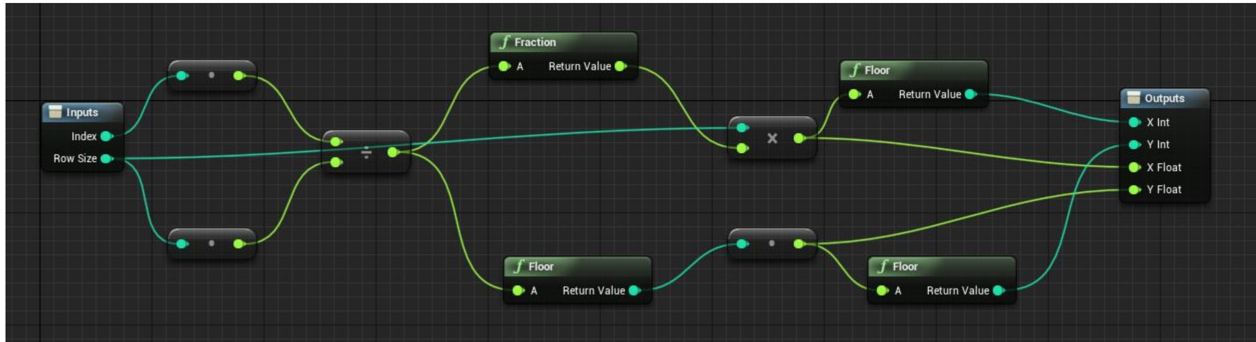


Figura 26: Conversión de entero a casilla.

Para ilustrar mejor el funcionamiento del código de la Figura 26 pondremos un ejemplo. Queremos averiguar en qué casilla (coordenadas x e y) está la celda número 15 en una matriz de 9x9. Si dividimos 15 entre 9 filas obtenemos 1.66666 periódico lo que nos indica que esa casilla está en la fila 2 (empezamos en 0). Ahora, usando la parte decimal de la división, la multiplicamos por el número de filas y obtenemos como resultado 6. Con esto obtenemos que la celda numero 15 corresponde a la fila 2 y columna 6 o lo que es lo mismo, x = 2 e y = 6.

6.1.4. Set initial tile

Una vez tenemos los bordes construidos es el momento de que definamos cual será nuestra casilla inicial dentro del laberinto. Esto es necesario, porque para construir el interior del laberinto, es necesario empezar por alguna casilla. En nuestro caso hemos decidido empezar por la casilla superior izquierda. Ver Figura 27.

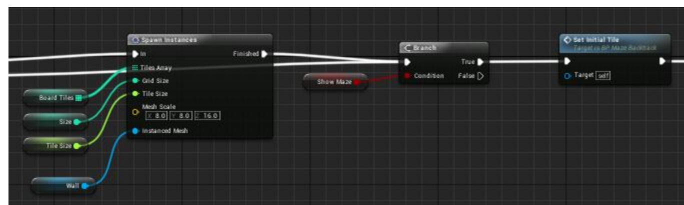


Figura 27: Definición de la casilla inicial.

En la Figura 28 podemos observar una nueva estructura de datos llamada ”path tiles”. Esta estructura servirá para almacenar las casillas del camino que estamos creando. Por tanto, la casilla inicial es la primera casilla de la estructura.

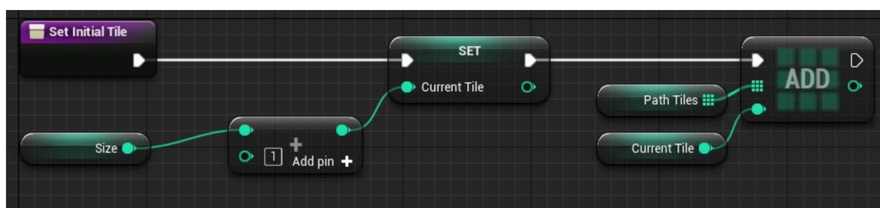


Figura 28: Función para designar la casilla inicial.

6.1.5. Bridges, pick nearby tile y backtrack

Hasta ahora hemos hecho las inicializaciones necesarias para construir el laberinto, ahora es el momento de la construcción de los caminos interiores. Para ello hemos denominado “bridges“ a los posibles caminos que podemos tomar.

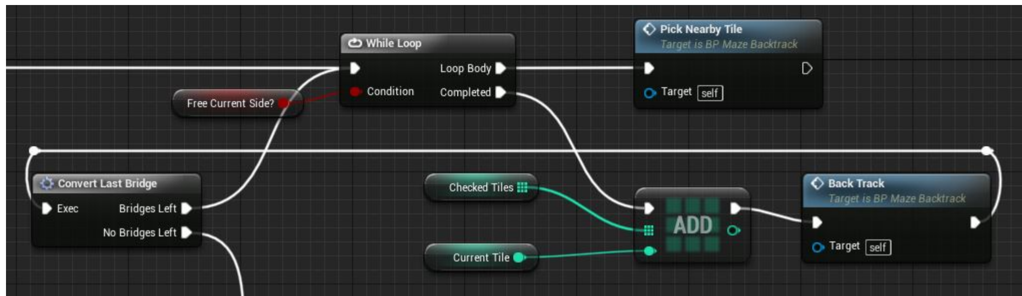


Figura 29: Bridges, pick nearby tile y bactrack.

En la Figura 29 podemos observar la variable boolean ”Free Current Side?”que indica si aún quedan caminos por explorar. Esta condición es la que controla el bucle que viene a continuación, donde ejecutaremos ”Pick Nearby Tile”.

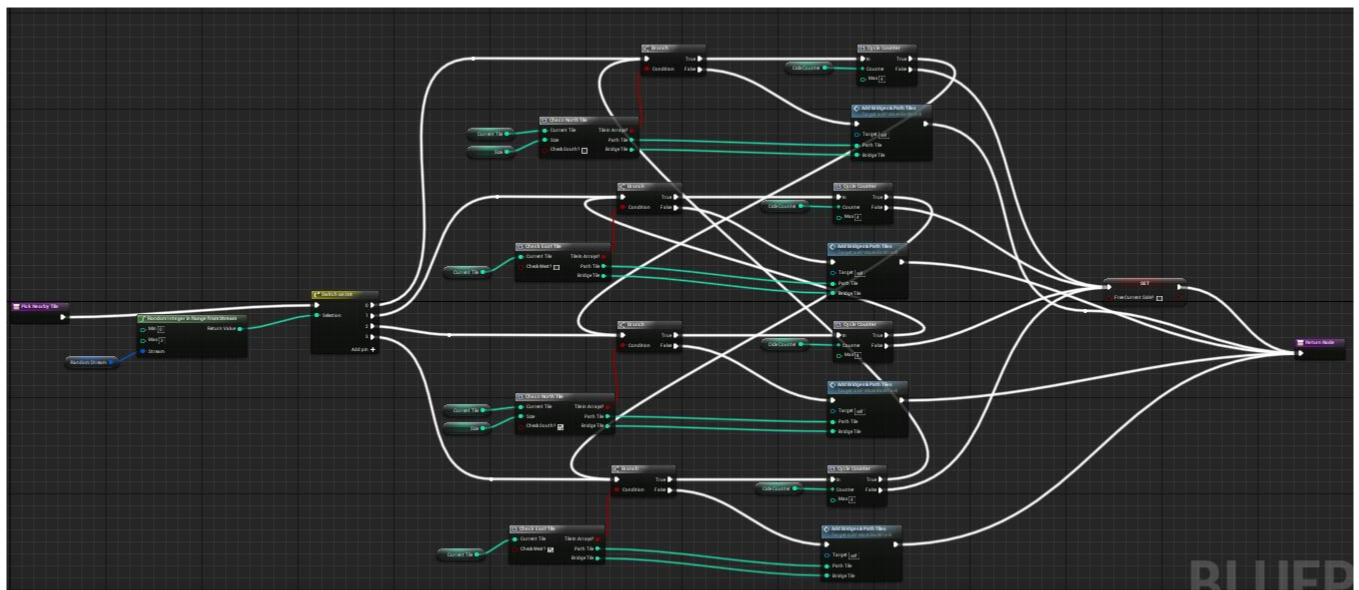


Figura 30: Visión global de Pick nearby tile. En las figuras siguientes la explicaremos paso a paso.

Esta función sirve para elegir las posibles casillas a las que podemos ir dentro del laberinto. La Figura 30 muestra la blueprint completa, pero a continuación vamos a desgranar su

contenido.

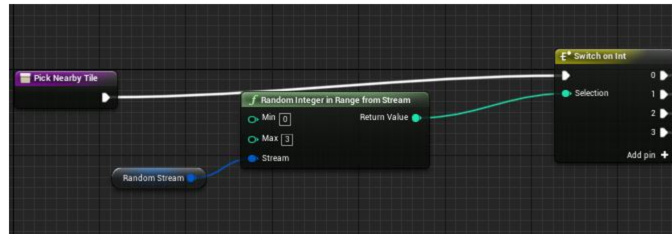


Figura 31: Inicio de Pick nearby tile.

Como observamos en la Figura 31, comenzamos con una semilla aleatoria que nos proporcionara un número aleatorio entre 0 y 3, ambos inclusive. Esto nos proporciona ese componente de aleatoriedad que nos ayudará a crear siempre laberintos distintos.

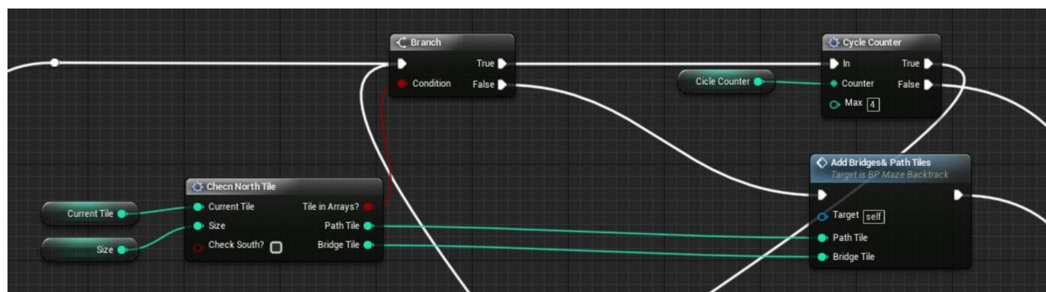


Figura 32: Check north tile.

El paso siguiente a lo visto en la Figura 31 es comprobar las distintas direcciones. Para ello hemos creado dos funciones llamadas "check north tile" y "check east tile". Estas funciones nos permiten mirar tanto una dirección como su contraria gracias al booleano de entrada. Aplicando las fórmulas mencionadas anteriormente obtendremos si la casilla que buscamos es accesible.

En la Figura 32 tambien observamos "Cycle counter" y "Add bridges & path tiles". Utilizamos Cycle counter para contar el número de lados que hemos comprobado (ver Figura 33) y Add bridges & path tiles añade la casilla bridge y la casilla del camino a las

correspondientes estructuras (ver Figura 34).

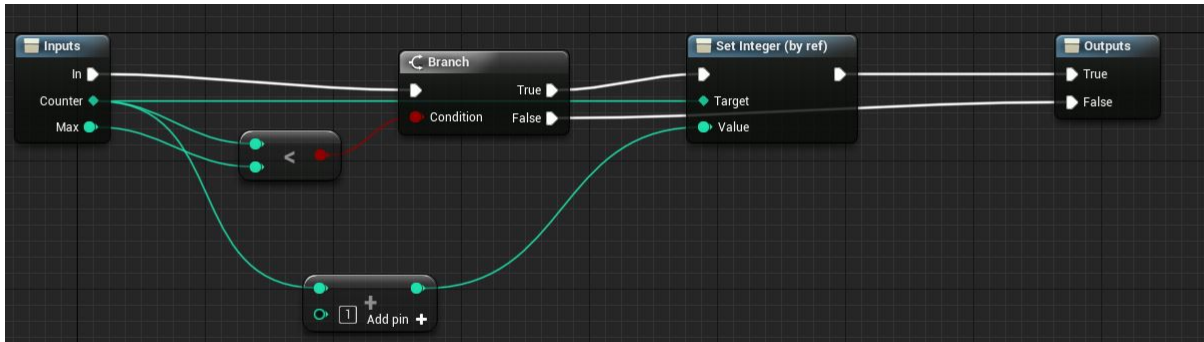


Figura 33: Cycle counter.

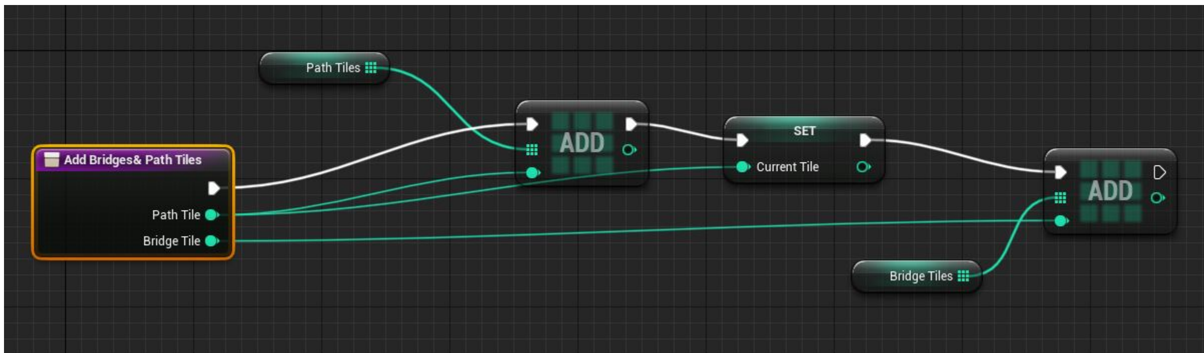


Figura 34: Add bridges & path tiles.

Check east tile comprueba bien la dirección este u oeste de la casilla actual para añadirla, igual que lo hace Check north tile, respectivamente.

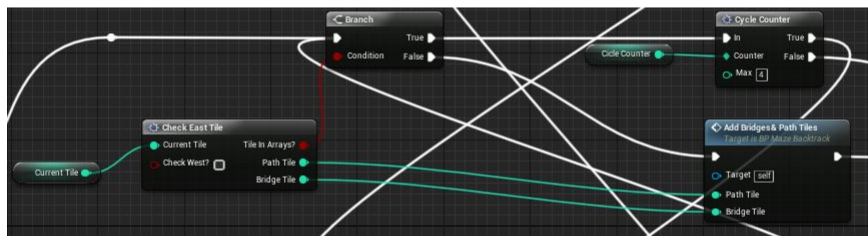


Figura 35: Check east tile.

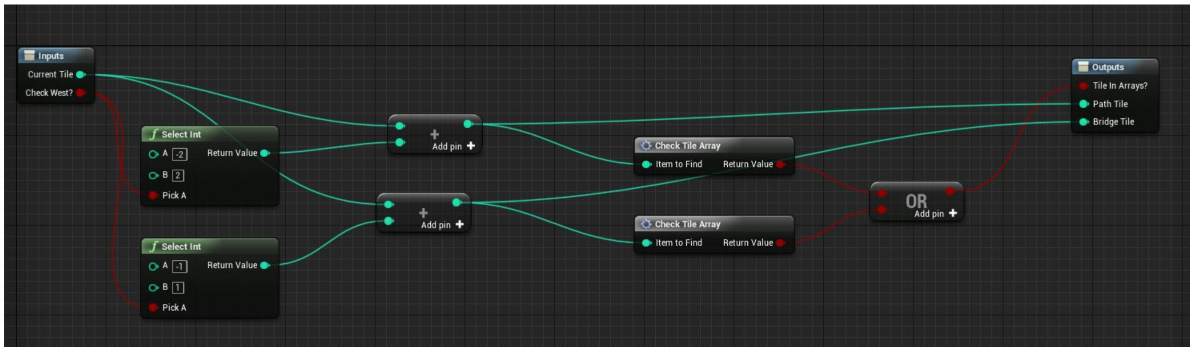


Figura 36: Funcion Check east tile.

Una vez hemos comprobado las 4 direcciones, como hemos mostrado antes, es hora de marcar que ya lo hemos hecho, cambiando el valor del booleano que controla el bucle (ver Figura 37).

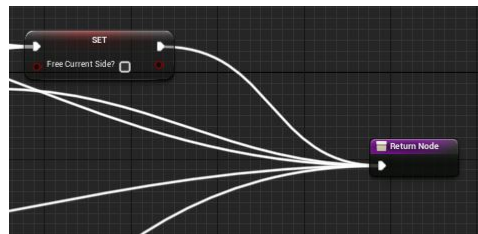


Figura 37: Final de Pick nearby tile.

Una vez escogidas las casillas candidatas, las añadimos a la estructura correspondiente y ejecutamos el bactrack (ver Figura 29). En esta función probaremos las distintas casillas candidatas para construir nuestro camino.

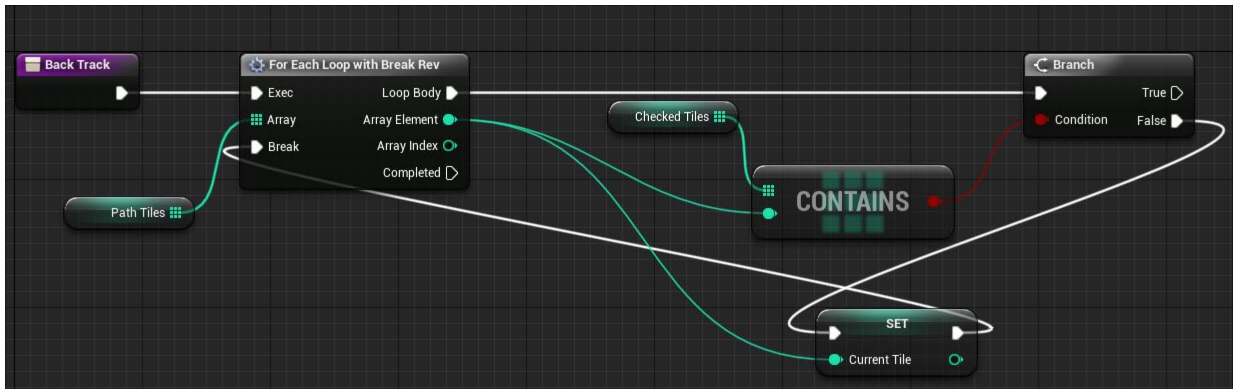


Figura 38: Función Backtrack.

A continuación de la función de backtrack ejecutaremos “Convert Last Bridge“ (ver Figura 29). Esta función enlaza con backtrack y entre ambas construyen el camino. Convert Last Bridge se encarga de comprobar si quedan casillas bridge por revisar. Si quedan, añade la última al camino y la marca como comprobada, además de quitarla de posibles candidatos. Por ultimo establece el booleano “Free Current Side“ a verdadero para que sigamos iterando (ver Figura 39).

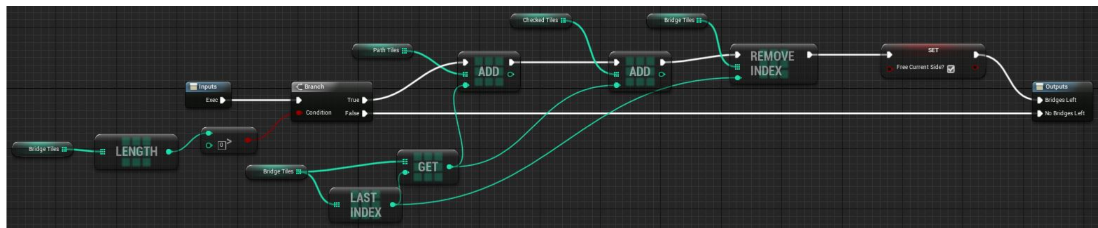


Figura 39: Función Convert Last Bridge.

6.1.6. Populate walls and loop paths

Una vez hemos definido los caminos dentro del laberinto, tenemos que identificar los muros interiores y podemos generar caminos en bucle o no. Ver Figura 40.

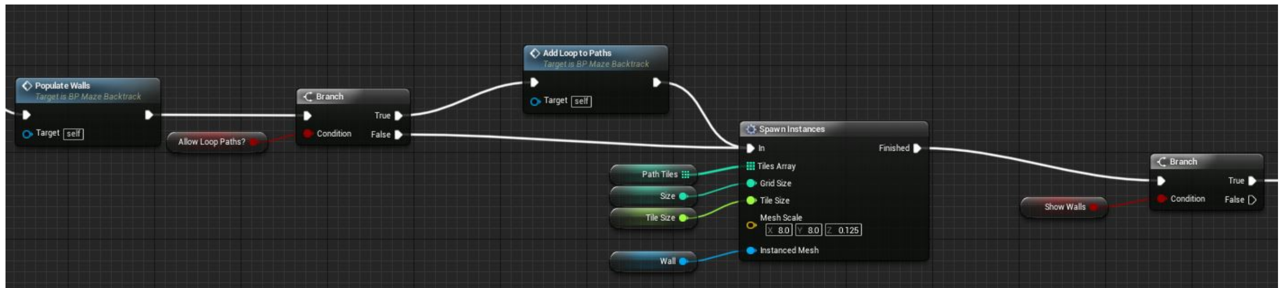


Figura 40: Populate walls and loop paths.

Los muros exteriores y caminos están definidos, por tanto,el resto de casillas son muros interiores. Para realizar este proceso hemos creado la función “populate walls“ que extrae el índice de estas casillas y lo guarda en una estructura distinta. Ver Figura 41.

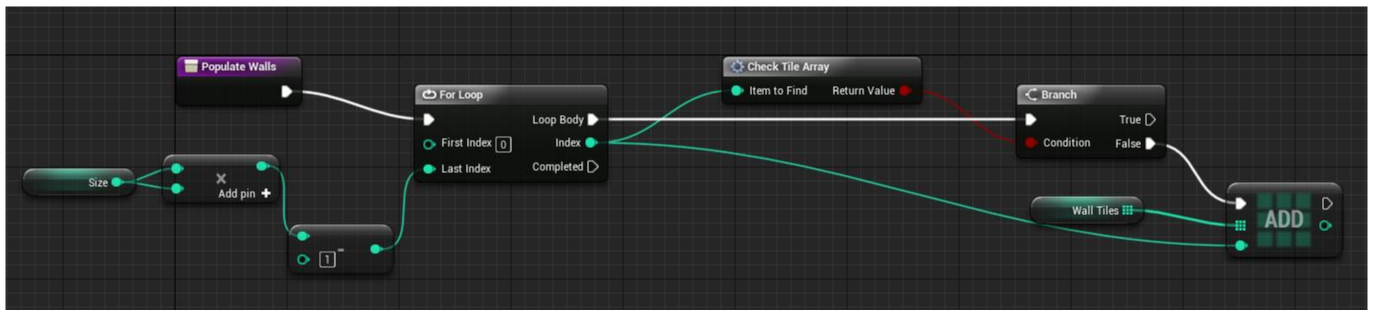


Figura 41: Función Populate walls.

Una vez tenemos los muros seleccionados podemos permitir que haya caminos en bucle o no gracias la condición. Ver Figura 40. Para crear estos caminos hemos creado la función “Add loops to path“. Ver Figura 42.

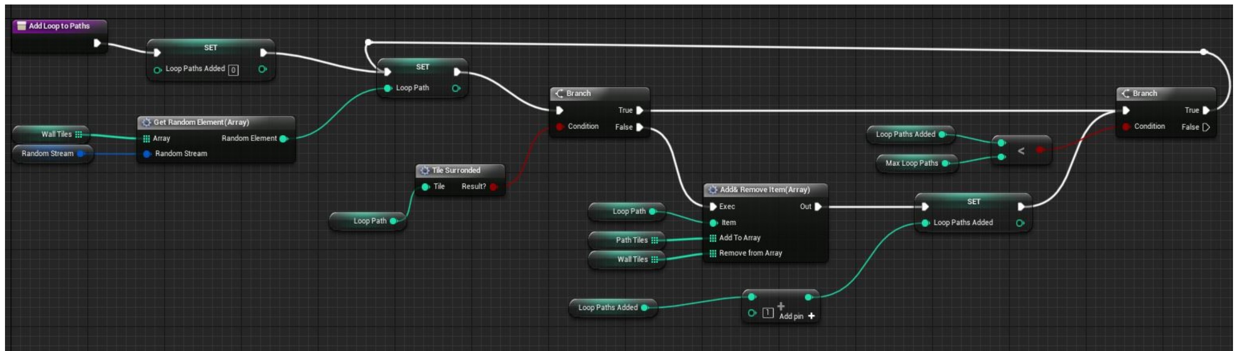


Figura 42: Función Add loops to path. En las Figuras siguientes las explicaremos paso a paso.

Para comenzar hemos de definir una función contar el número de caminos en bucle, y definir un máximo. Para crear un camino en bucle lo que haremos será seleccionar un muro interior aleatorio y a partir de él crear nuestro camino en bucle. Ver Figura 43.

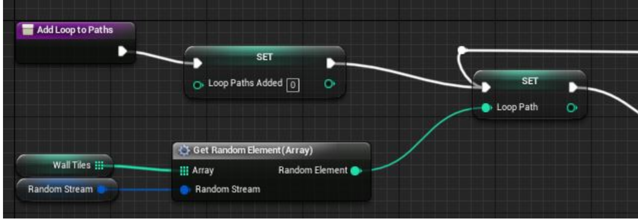


Figura 43: Inicio de la función Add loops to path.

Una vez hemos realizado la anterior y si tenemos seleccionado nuestro muro, hemos de comprobar si quitándolo formamos un camino que no sea en bucle. La función "Tile surronded" es la que se encargar de ello. Más adelante explicaremos como lo hace (ver Figura 45). Si el resultado es falso, es decir, que el camino es un bucle, lo que hacemos es eliminar esa casilla de los muros y añadirla a los caminos, aumentando así el número de caminos en bucle que tenemos. Tanto si añadimos un nuevo camino como si no, debemos comprobar si tenemos ya el número máximo de caminos, si es que no seguiremos añadiéndolos, y si tenemos el máximo termina la función. Ver Figura 44.

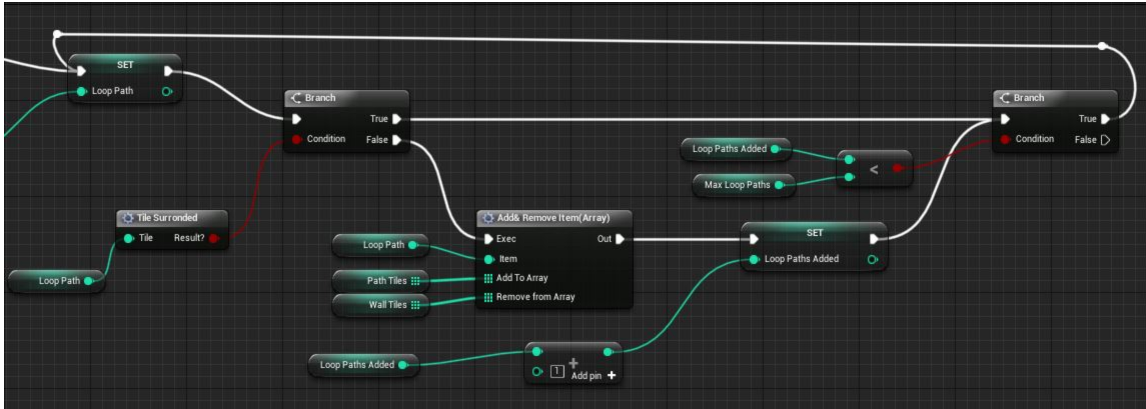


Figura 44: Fin de la función Add loops to path.

Para determinar si el camino es un bucle o no lo es, hemos creado la función “Tile surronded“. Para que una casilla sea apta para formar un camino en bucle debe estar al lado de dos caminos opuestos (Norte y sur, Este y Oeste). La razón por la que hacemos esto es para evitar que, si la casilla seleccionada es el final de un muro, se formen espacios de 2x2. Lo primero que hacemos es obtener el número de la casilla correspondiente dentro de la matriz, después comprobamos si esas casillas están en los caminos creados previamente y, por último, comprobamos si la casilla es apta según el criterio anterior. Ver Figura 45.

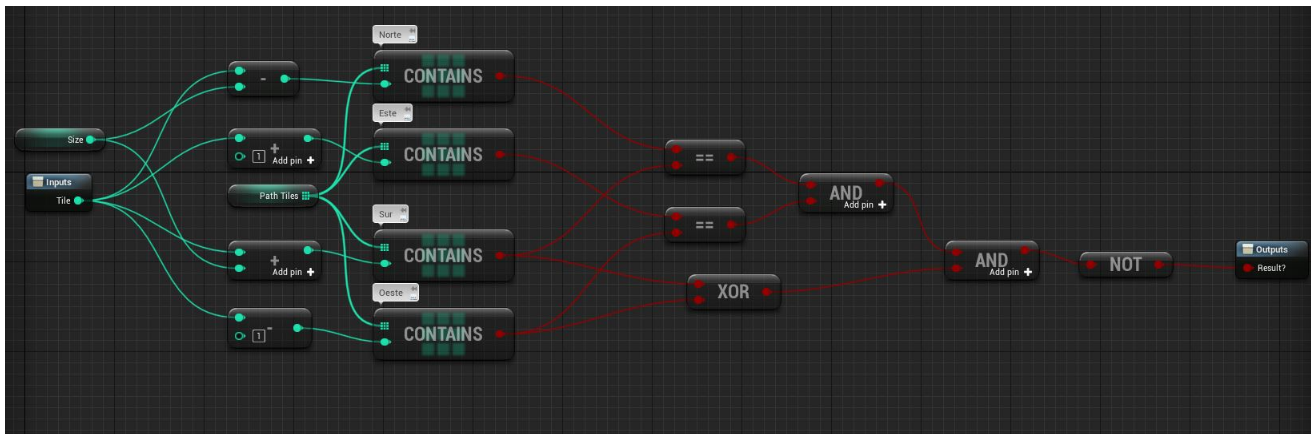


Figura 45: Función Tile surronded.

Tras todo esto, el último paso es crear los caminos definitivos con o sin bucles. Ver Figura 46.

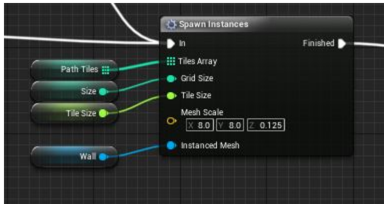


Figura 46: Creación de los caminos.

Para dotar al laberinto de una mayor variedad hemos añadido la función que los muros puedan tener una altura aleatoria. Esto le proporciona una mayor sensación de distinción con otros laberintos, aunque es meramente estético. Creamos este efecto escalando el modelo en el eje Z y generando un numero aleatorio en vez de usar un valor fijo. Una vez hecho esto, creamos todos los muros interiores. Ver Figura 47.

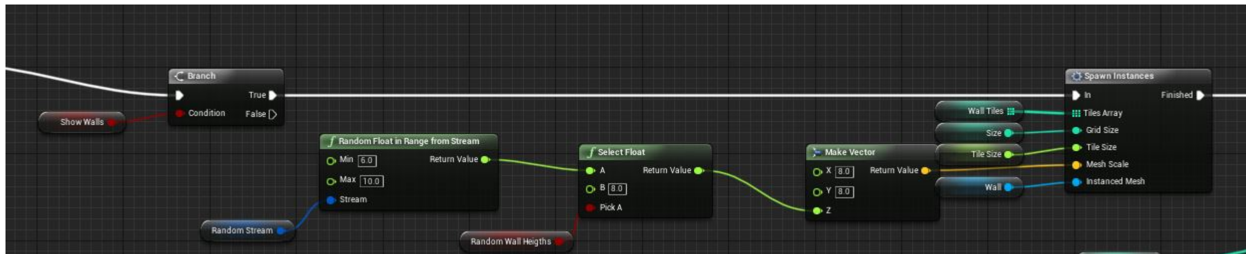


Figura 47: Creación de muros con altura aleatoria.

6.2. Sistema de misiones

Para hablar del sistema de misiones, primero hemos de hablar sobre las estructuras básicas de datos que hemos construido. Nuestro sistema está construido para soportar fácilmente los cambios en los datos. Para ello hemos creado unos archivos CSV (Comma Separate Values)

que almacenan los datos. Estos archivos son editables desde fuera de Unreal Engine con cualquier editor de texto. Una vez cargados en el sistema, estos datos son asignados a su correspondiente objeto dentro del sistema gracias a su nombre que actúa como identificador único.

Para que el sistema de misiones funcione correctamente necesitamos más elementos que lo hagan posible. Estos elementos son:

- **Diálogos** donde el texto sea variable y permita que el jugador tome decisiones. Ver 6.3.5.
- **Inventario** donde el jugador guarde los objetos recogidos para usarlos o entregarlos para completar misiones. Ver 6.4.
- Un **menú de interacción** con los distintos NPC y los objetos para que jugador pueda hablar, recoger, etc. Ver 6.5.
- **Objetos interactuables** para que el jugador pueda recogerlos, inspeccionarlos, etc. Ver 6.6.

Todos estos elementos los explicaremos en las próximas secciones del documento.

6.2.1. Estructuras de datos

Para los NPC (Non Playable Character) hemos creado una estructura con su nombre y un texto de presentación del propio personaje. Ver Figura 48.

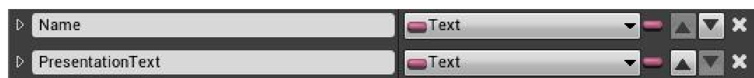


Figura 48: Estructura de datos para los NPC.

Los objetos que los jugadores pueden recoger estarán repartidos por todo el escenario y son identificables por el nombre, pero puede haber varios objetos con el mismo nombre. Los objetos tienen los siguientes atributos (ver Figura 49):

- **Name**: nombre que identifica el objeto.
- **Description**: descripción del objeto cuando es inspeccionado.
- **Combinable**: si el objeto es resultado de una combinación de otros objetos.
- **Item1**: el primero objeto de la combinación, si el objeto es combinable.
- **Item2**: el segundo objeto de la combinación, si el objeto es combinable.
- **Stackeable**: si el objeto en el inventario puede tener agrupaciones con objetos iguales.
- **Used**: si el objeto esta usado.
- **Thumbnail**: la imagen que usaremos para mostrarlo en el inventario.

Figura 49: Estructura de datos para los objetos.

Hemos creado una estructura para las misiones. Esta estructura no obtiene los datos de un archivo, si no que la usamos para crearlas en función de los datos leídos anteriormente. Las misiones tienen los siguientes atributos (ver Figura 50):

- **QuestType**: esto identifica el tipo de misión, puede ser 0 para una misión de recoger objetos y un 1 para una misión de diálogo.
- **NPC**: el personaje que entrega la misión a los jugadores.
- **Item**: el objeto que el jugador debe entregarle al NPC en caso de ser una misión de recoger objetos.
- **NPCToTalk**: el NPC con el que hay que dialogar para completar una misión de diálogo.

Figura 50: Estructura de datos para las misiones.

6.2.2. Creación de misiones

El sistema de misiones que hemos creado se divide en bloques, estos bloques tienen como mínimo 1 misión y como máximo 3. Este número es aleatorio. Ver Figura 51.

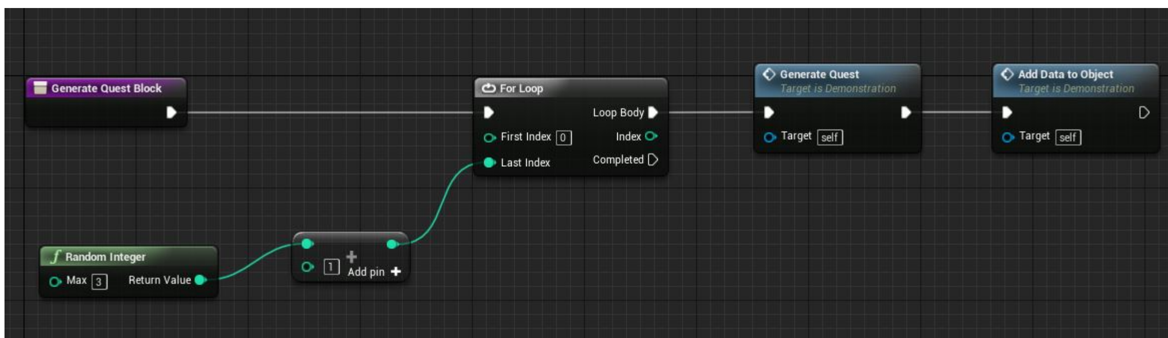


Figura 51: Generar un bloque de misiones.

Por último, para generar nuestro bloque de misiones, lo que hacemos es añadir los datos obtenidos del archivo a los objetos. Para ello, compararemos el nombre de los objetos con los del archivo y si coinciden le añadimos los datos. Ver Figuras 52 y 53.

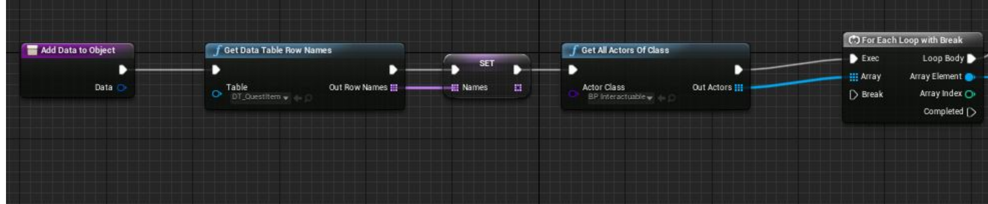


Figura 52: Obtener objetos e iterar sobre ellos.

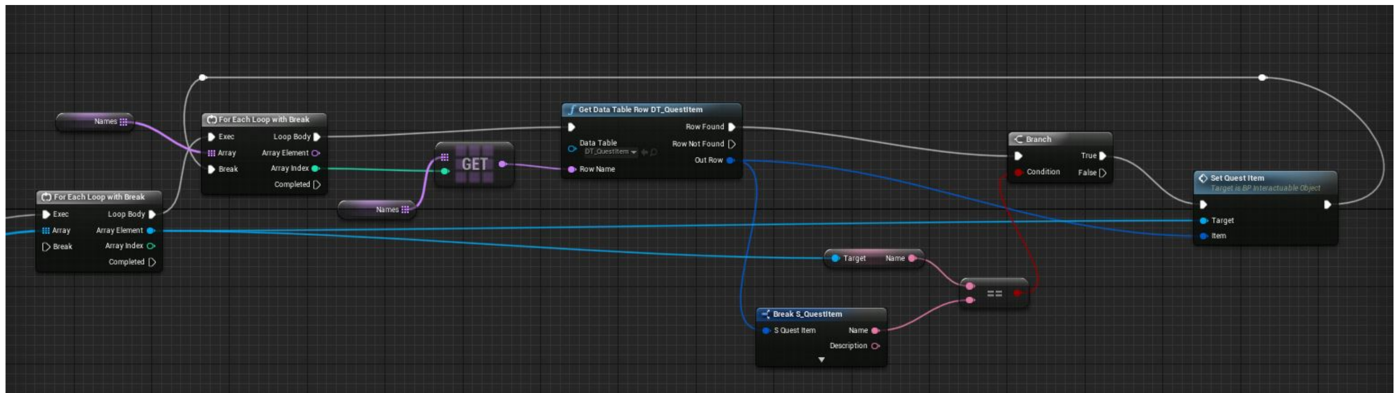


Figura 53: Obtener datos del archivo,iterar y, si coinciden, los nombres añadir los datos.

Por ejemplo, si cambiamos o añadimos algún dato del archivo no sería necesario cambiar nada de la implementación. Tan solo cargar la última versión del archivo al sistema y ya tendríamos la actualización hecha.

Para crear las misiones según la estructura mostrada anteriormente (ver Figura 50) hemos creado un número aleatorio que decide el tipo de misión que generaremos, ademas, elegimos un NPC aleatorio.

Si el número generado es 0 entonces sera una misión de recolección. Por lo tanto obtene-

mos un objeto aleatorio, creamos nuestra misión y se la añadimos al NPC que seleccionamos en primer lugar. Por último, añadimos los datos obtenidos del archivo al objeto en el escenario.

Si el número generado es 1 entonces sera una misión de diálogo. Por lo tanto obtenemos un NPC aleatorio distinto del primero, creamos nuestra misión y se la añadimos al NPC que seleccionamos en primer lugar y al NPC que obtuvimos en segundo lugar. Ver Figura 54.

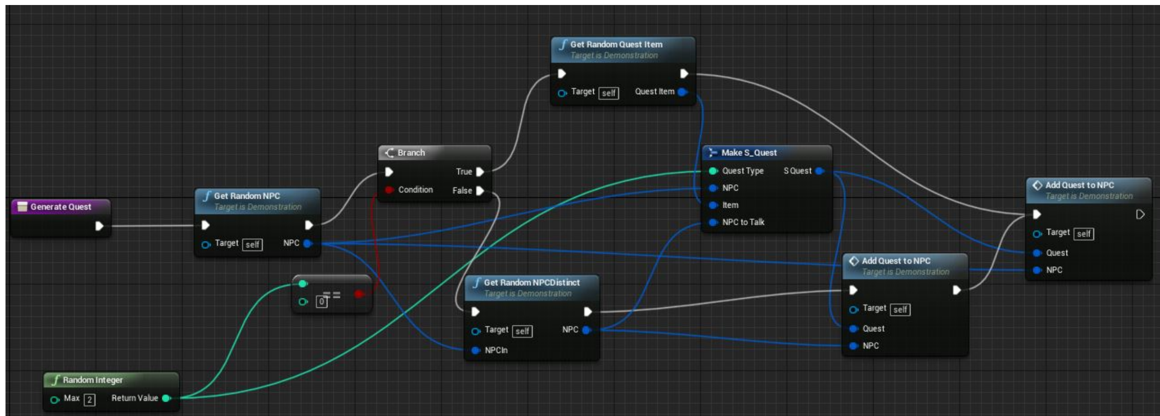


Figura 54: Generar una misión. En las Figuras siguientes lo explicaremos paso a paso.

Para obtener un NPC aleatorio, lo primero que hacemos es recuperar los datos del archivo en el que están los NPC. Una vez hecho esto, generamos un número aleatorio entre 0 y el numero de registros en la tabla de datos de los NPC. Entonces, el número que se genere corresponderá a un registro dentro de la tabla, el cual será devuelto. Ver Figura 55.

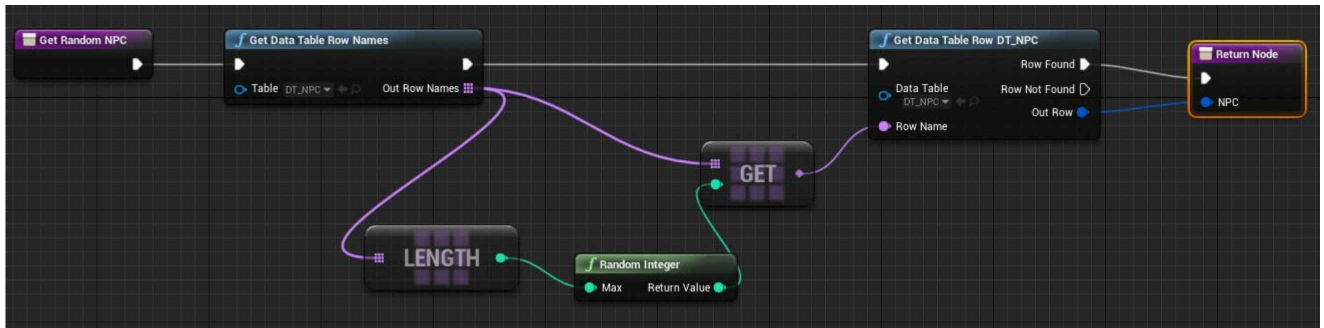


Figura 55: Obtener un NPC aleatorio.

El procedimiento para obtener un objeto aleatorio es el mismo que para los NPC, pero usando la tabla de objetos. Ver Figura 82.

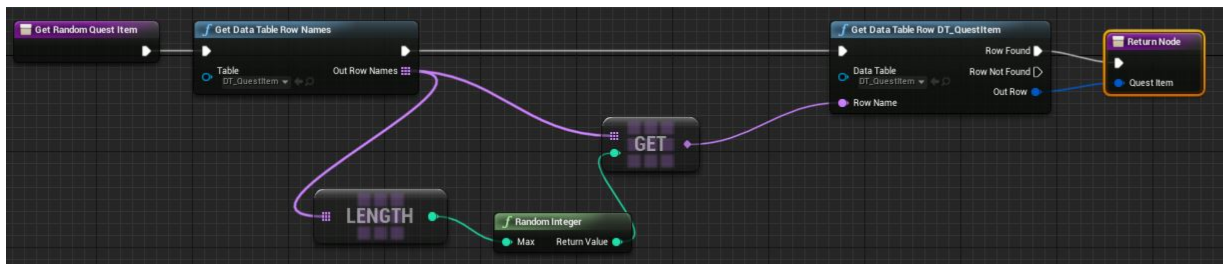


Figura 56: Obtener un objeto aleatorio.

Una vez tenemos esto, crearemos nuestra misión y se la añadiremos al NPC correspondiente. Para ello, lo que haremos será buscar qué NPC es al que hay que asignársela. Lo obtendremos iterando a través de todos los actores del tipo NPC que tengamos y cuando coincida el nombre con el que estamos buscando, añadimos la misión a su estructura que contiene todas las misiones que le pertenecen. Ver Figura 57.

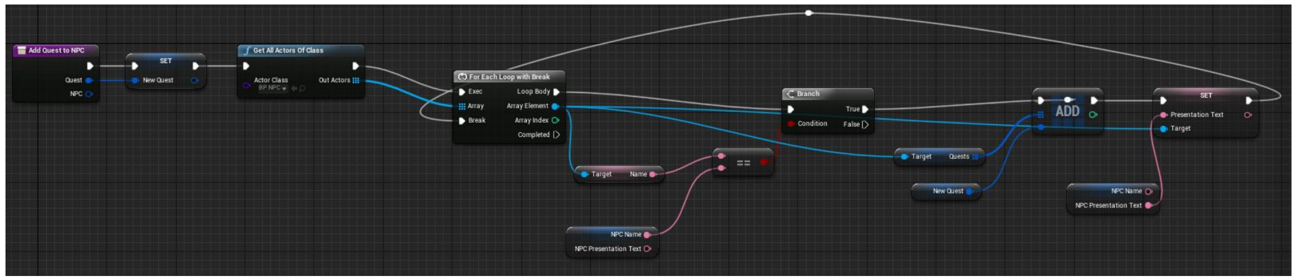


Figura 57: Asignar misión a un NPC.

Si la misión es del tipo diálogo en vez de seleccionar un objeto, lo que haremos será seleccionar un NPC aleatorio distinto del primero. Lo haremos seleccionando uno aleatorio y comprobando que no sea el mismo que el NPC que recibe la función. Ver Figura 58.

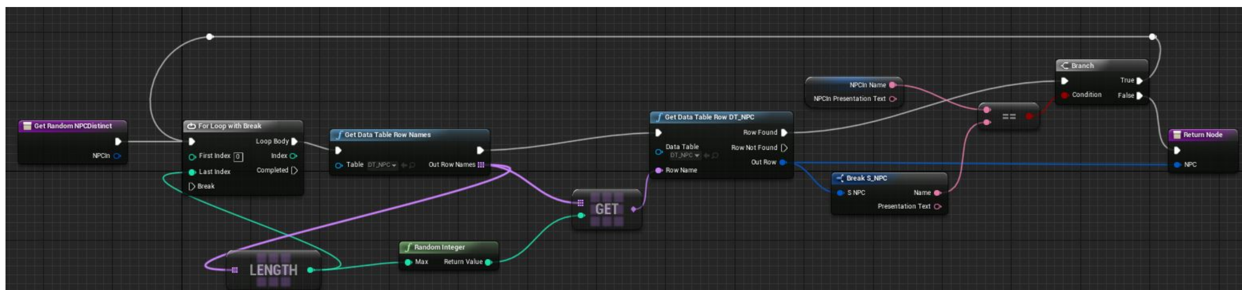


Figura 58: Obtener un NPC aleatorio distinto del primero.

Por ejemplo, tenemos los NPC llamados "A", "B" y "C". El primer NPC elegido es "B" (registro número 1 en la tabla), al generar un numero aleatorio obtenemos el número 1 que coincide con el registro que ya tenemos. Al comprobar si los nombres son iguales veremos que lo son. Entonces haremos de nuevo el mismo proceso hasta obtener otro NPC distinto, que puede ser 'A" o "C".

6.3. Arboles de diálogo

Para comenzar esta sección debemos explicar qué son los **Behaviour Trees** en Unreal Engine. Un Behaviour Tree contiene una representación gráfica en forma de árbol de una lógica a base de condiciones. En Unreal Engine se suelen utilizar para la programación de la inteligencia artificial. Otro concepto que aparece en los Behaviour Trees en Unreal Engine es el Blackboard. El Blackboard contiene información del estado del juego que es compartida a todo el Behaviour Tree y le ayuda a tomar las distintas decisiones.

6.3.1. Behaviour Tree

En este proyecto lo hemos utilizado para implementar las distintas opciones de diálogo que puede considerar un NPC. Antes de explicar la implementación, es necesario conocer algunos nodos que usa Unreal Engine en los Behaviour Trees. Los nodos son los siguientes:

- **Root**: este es el nodo raiz presente en todos los Behaviour Trees. En él se almacenan los datos del Blackboard.
- **Sequence**: ejecuta sus nodos hijos de izquierda a derecha hasta que alguno falla. Si falla un hijo, entonces se considera que el nodo sequence también falla.
- **Selector**: ejecuta sus nodos hijos de izquierda a derecha hasta que alguno se ejecuta con éxito.
- **Task node**(nodo morado): representan la realización de alguna acción dentro del árbol.
- **Decorator** (nodo azul): son un tipo especial de nodos que añaden funcionalidades a los nombrados anteriormente. Esta funcionalidad por lo general suele ser una condición para la ejecución de sus hijos.

Gracias a estos nodos construiremos un árbol de diálogo que se adapte a todas las posibles misiones de ese tipo. Por ejemplo, si tenemos una misión de tipo diálogo generada, no es lo mismo que una misión de recogida de objetos, por lo tanto, los árboles de diálogo tampoco pueden ser iguales (ver Figura 54).

6.3.2. Blackboard

Para poder construir los Behaviour Trees hemos utilizado el Blackboard para compartir la información entre los distintos nodos. El blackboard es el mismo para todos los árboles y contiene las siguientes variables (ver Figura 59):

- **DialogueWidge**t: es el objeto que muestra los diálogos en pantalla.
- **SelfActor**: el actor al que pertenece el Behaviour Tree.
- **Response**: la respuesta que da el jugador al NPC.
- **WaitingObject**: si el NPC espera un objeto para completar la misión.

Figura 59: Variables en el Blackboard.

6.3.3. Task Nodes implementados

Los Task Nodes son nodos que ejecutan una **tarea**. Lamentablemente, los que Unreal Engine tenía implementados no nos eran útiles por lo que implementamos nuestros propios

nodos. Por lo general, los Task Nodes de Unreal Engine son para mover el NPC, que ataque o que realice una acción en el escenario. Para nuestro sistema necesitábamos algo que interactuara con la interfaz y con algunas de las variables de nuestro sistema, por lo que decidimos implementar nuestros propios nodos.

Para iniciar los diálogos creamos un nodo que realiza todas las tareas necesarias en la interfaz, llamado T_StartDialogue. Lo que hace es obtener el NPC controlado y obtener el modo de juego para poder luego acceder a la interfaz. Ver Figura 60.

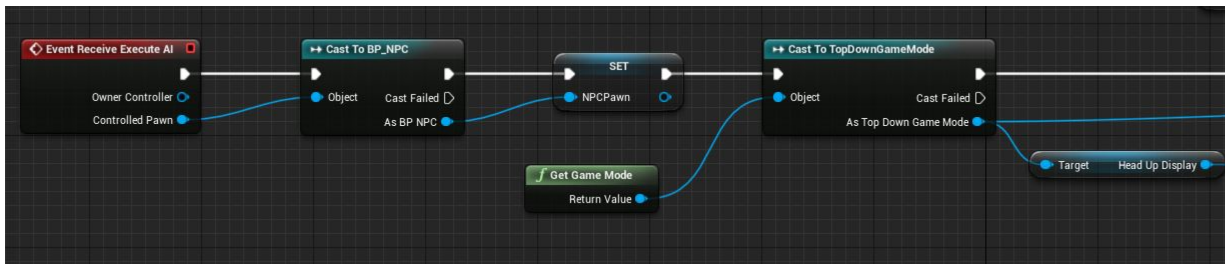


Figura 60: Inicio del nodo de inicio de diálogo.

Una vez tenemos acceso a la interfaz, debemos poner el nombre del personaje que habla en la propia interfaz, hacerla visible y por último ocultar el menú de interacción. Ver Figura 61.

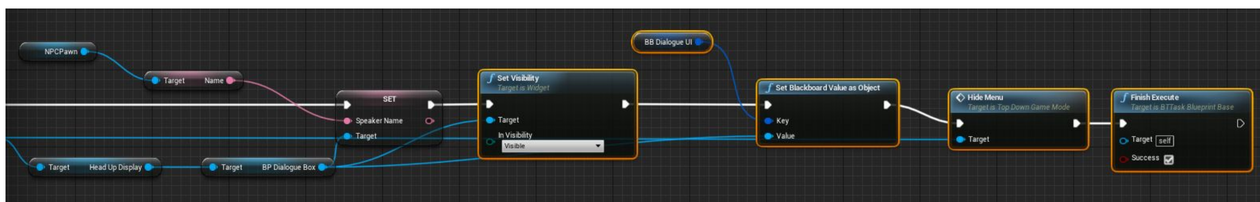


Figura 61: Final del nodo de inicio de diálogo.

Para mostrar lineas de diálogo utilizamos el nodo T_DisplayLine. Con este nodo lo que hacemos es asignar el texto a la interfaz y esperamos que el jugador puse el botón derecho

del ratón para pasar a la siguiente linea. Ver Figuras 62 y 63.

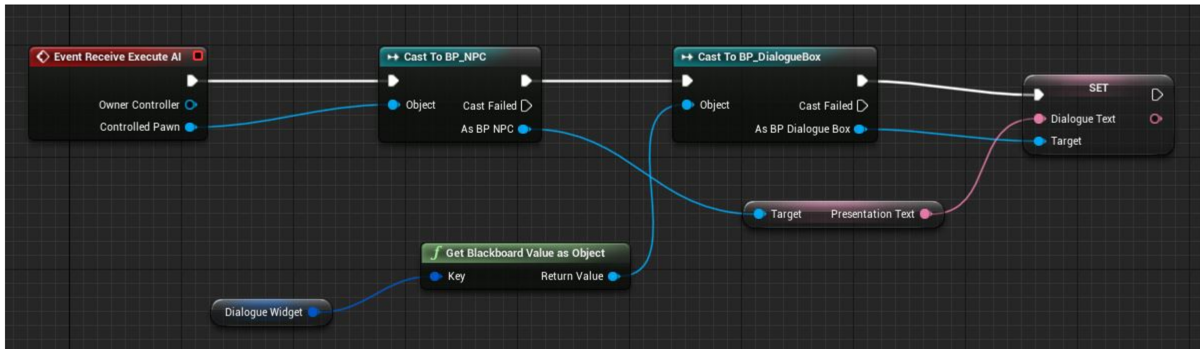


Figura 62: Asignación del texto.

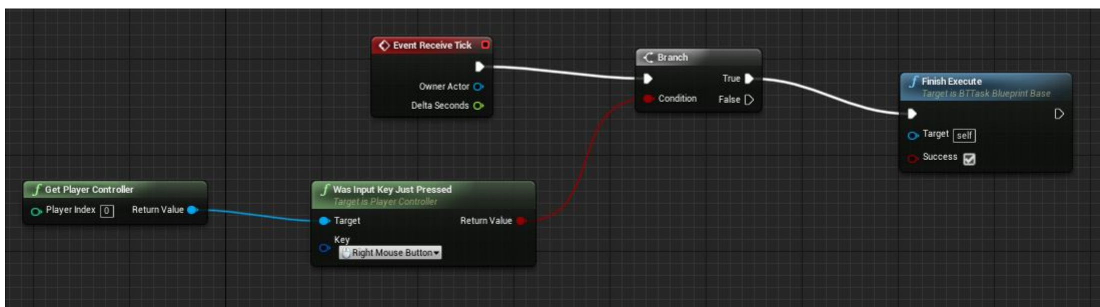


Figura 63: Controlador para cambiar de linea de texto.

Hay varias partes textuales que se mostraran bastante a menudo, por lo que hemos creado un Task Node específico para ellas.

Para mostrar el mensaje adecuado cuando un NPC espera un objeto utilizamos el nodo T_DisplayWaitMessage. Mostramos un mensaje para indicar que el NPC espera el objeto indicado (obtenemos el nombre del objeto de la misión activa) y esperamos que el jugador puse el botón derecho del ratón para pasar a la siguiente línea. Ver Figuras 63 y 64 .

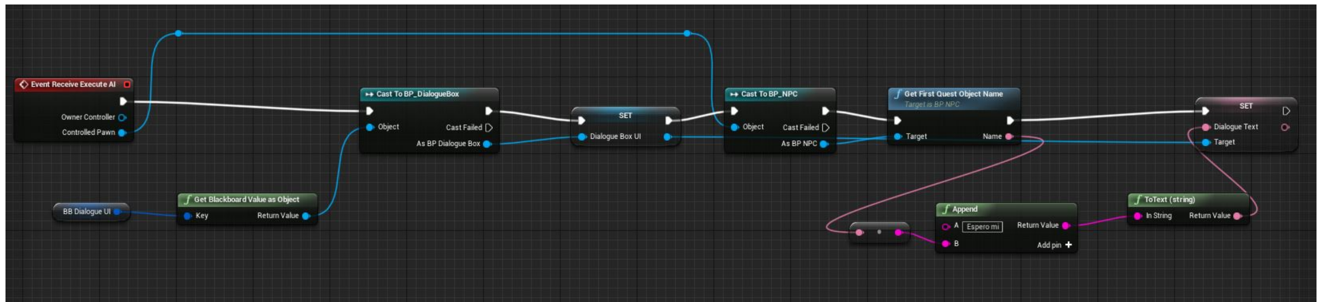


Figura 64: Mostrar el mensaje de espera según el objeto.

Para mostrar la presentación de un NPC utilizamos el nodo T_DisplayPresentation. Mostramos la presentación del NPC y esperamos que el jugador puse el botón derecho del ratón para pasar a la siguiente línea. Ver Figuras 63 y 65.

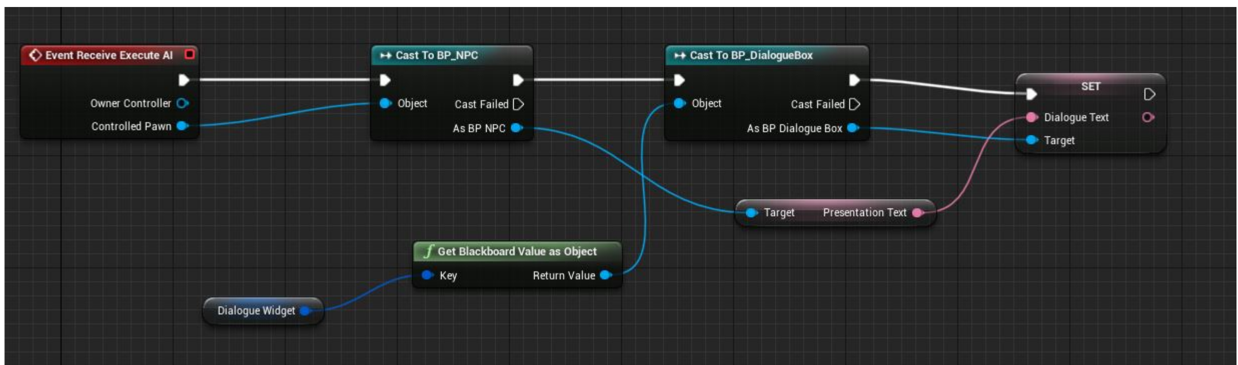


Figura 65: Mostrar presentación.

Pedir al jugador un objeto es algo que se repite con mucha frecuencia en el proyecto y para ello hemos creado el nodo T_AskQuestion. En él establecemos el texto que se muestra en con el nombre del objeto deseado y preparamos las respuestas para mostrarlas en la interfaz. Ver Figura 66.

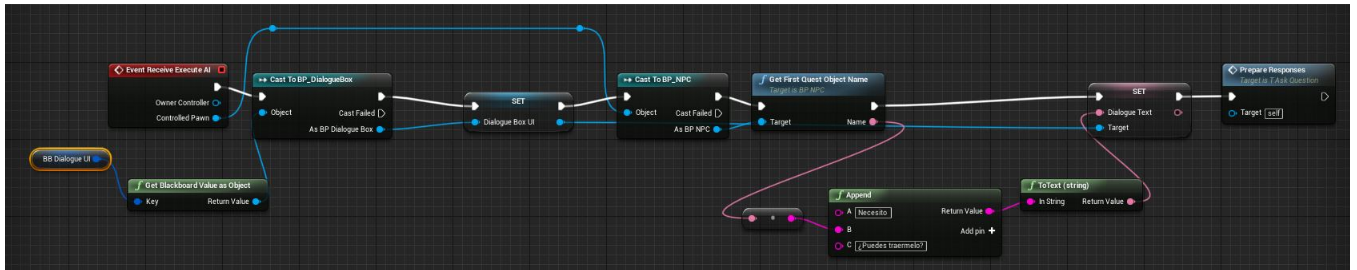


Figura 66: Mostrar el mensaje de espera según el objeto.

Para mostrar las posibles respuestas usamos la función “Prepare Responses“. Primero hemos de crear la interfaz que se muestra en pantalla (Response List) y añadirla a la interfaz. Ver Figura 67.

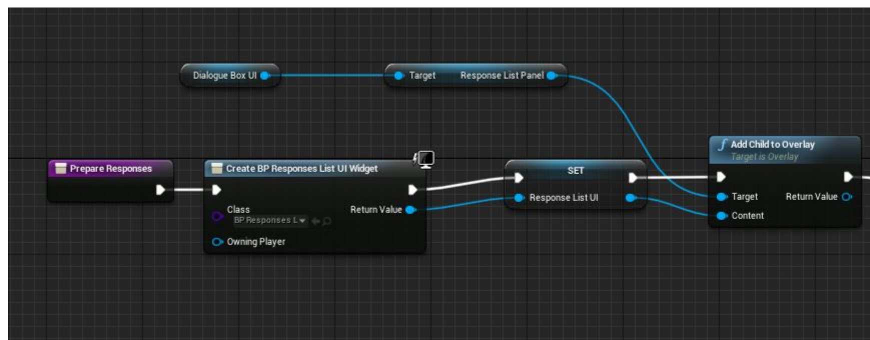


Figura 67: Creando la Response List.

Una vez tenemos la lista de respuesta añadida a la interfaz, creamos por cada posible respuesta un botón y lo añadimos a la “Response List“. Por último, asociamos un evento a los botones para saber cuando el jugador ha pulsado una respuesta. Ver Figura 68.

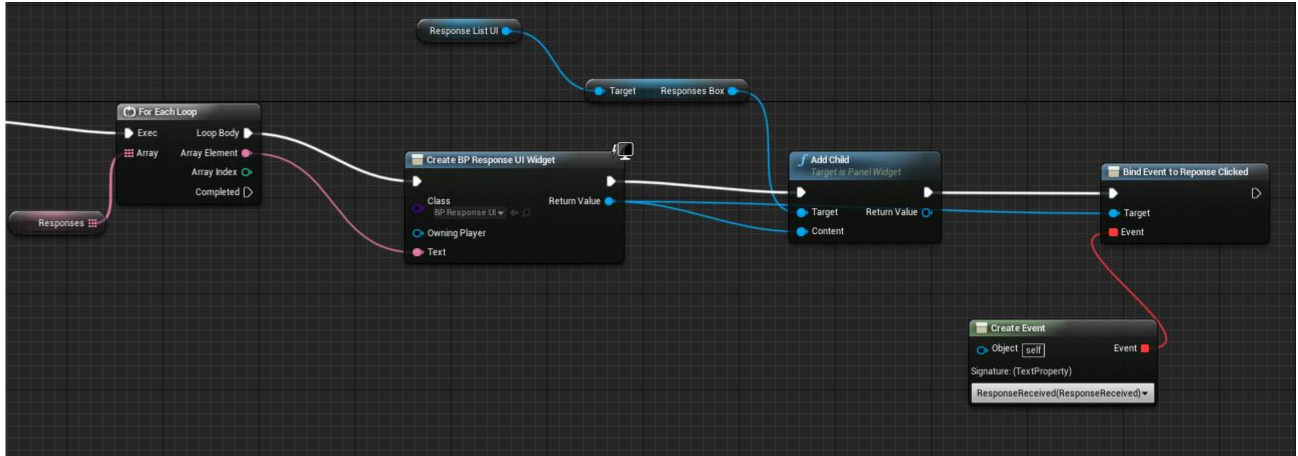


Figura 68: Mostrar respuestas.

Cuando el jugador responde se lanza un evento que asigna la respuesta en el Blackboard y quita las respuestas de la interfaz. Ver Figura 69.

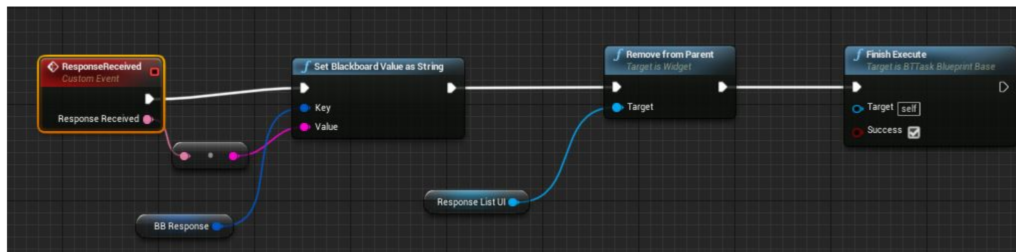


Figura 69: Evento de respuesta.

Esta manera de hacer las preguntas es la misma que utilizamos para las misiones de diálogo, tan solo cambiamos el nombre del objeto por el del NPC y el texto mostrado en la pregunta. Este Task Node se llama T_AskTalkQuestion. La diferencia la podemos ver en la Figura 66 y la Figura 70.

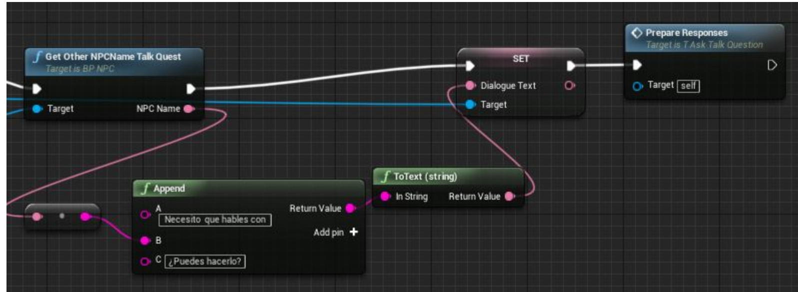


Figura 70: Diferencia con el nodo T_AskQuestion.

Si el jugador acepta la misión, debemos marcarla como activa. Lo hacemos con el Task node llamado T_SetActiveQuest. Ver Figura 71.

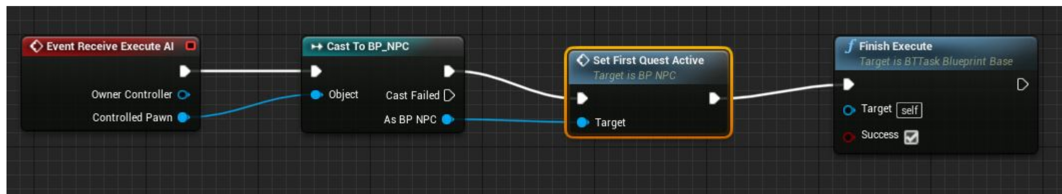


Figura 71: Marcar misión activa.

Para marcar la misión como activa, tan solo tenemos que obtener la primera misión que tiene el NPC y poner un booleano a True. Ver Figura 72.

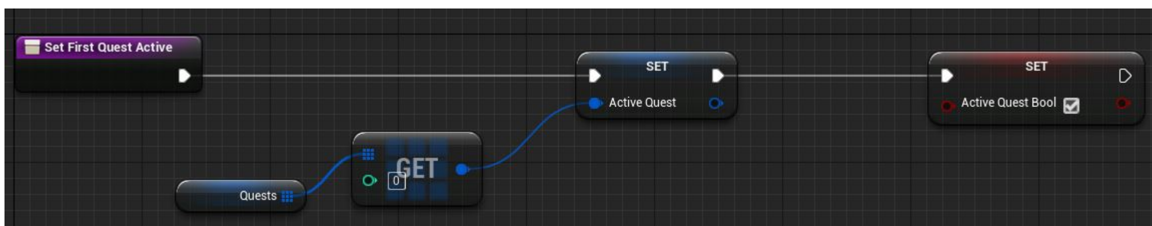


Figura 72: Marcar primera misión activa.

Cuando se trata de una misión de diálogo, el procedimiento es el mismo, pero además de activar la misión en el NPC que la entrega, también debemos hacerlo en el que es objetivo de

la misión. Para hacerlo marcaremos la primera misión de ese NPC como activa (ver Figura 72). Ver Figuras 73 y 74.

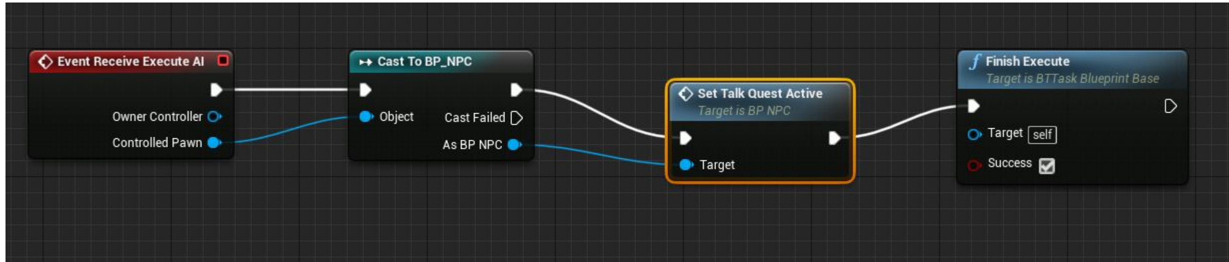


Figura 73: Marcar misión de diálogo como activa.

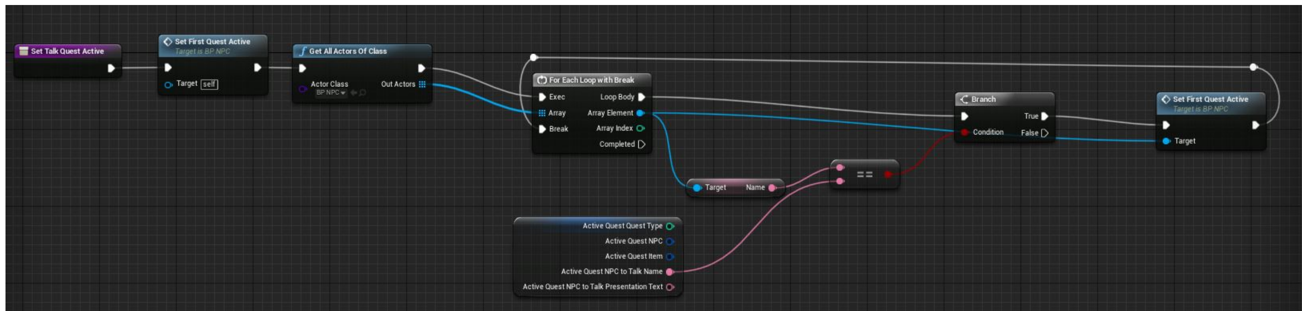


Figura 74: Activando misiones de diálogo en ambos NPC.

Para finalizar el diálogo usamos el nodo T_EndDialogue. Este Task Node oculta la interfaz de diálogo de la pantalla dado que no la usaremos más. A continuación, para la lógica de ejecución del NPC, parando así la ejecución del árbol (Figura 76) y permite que el jugador se mueva de nuevo cambiando el valor del booleano. Ver Figura 75.

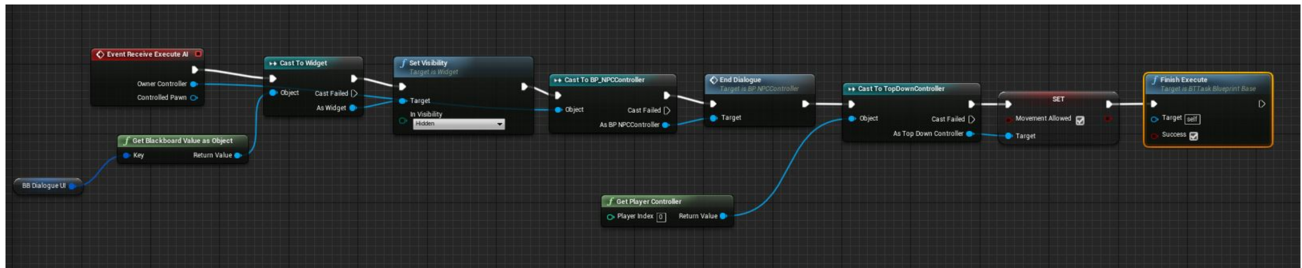


Figura 75: Finalizando el diálogo.

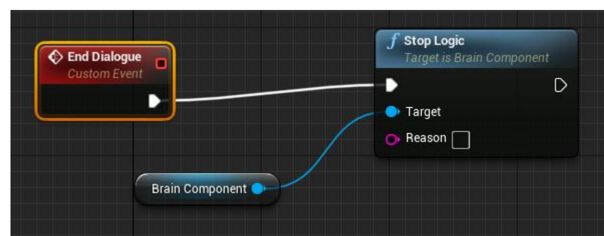


Figura 76: Finalizando lógica del NPC.

Todos estos nodos mostrados se pueden considerar acciones atómicas. Combinando todas estas acciones podemos construir **árboles** que satisfagan todas las necesidades para poder completar las misiones. Por ejemplo, el jugador tiene una misión de recoger una botella. En este ejemplo, el jugador habla con el NPC al que debe entregarle la botella, el NPC debe comprobar si el jugado tiene en su inventario la botella y, si la tiene, eliminarla. Si el jugador no la tuviera, el NPC no puede eliminar la botella ni completar la misión. Ver Figura 86

6.3.4. Decorators implementados

Para la comprobación de condiciones hemos implementado unos nodos de tipo Decorator que nos ayudan a seleccionar las acciones que ejecutan los NPC. Para elegir la frase del NPC tenemos que saber si tiene una misión activa o no. El nodo D_CheckActiveQuest comprueba el valor de esa variable en el NPC. Ver Figura 77.

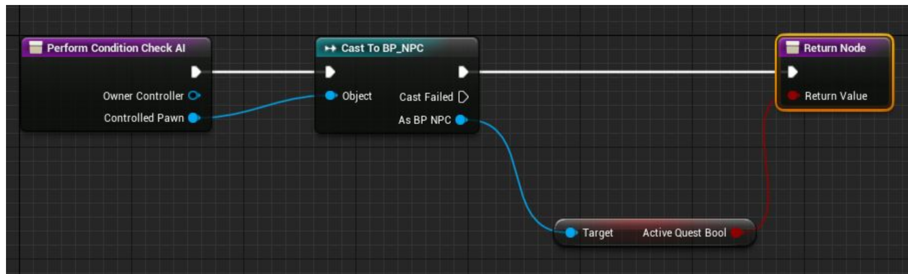


Figura 77: Comprobación de una misión activa.

Cuando el jugador responde una pregunta debemos comprobar qué respuesta es, para que el flujo tome una ruta u otra dentro del árbol. El nodo D_CheckActiveQuest revisa la respuesta dada por el jugador y almacenada en el Blackboard para comprobar si es la misma para seguir por esa rama del árbol. Ver Figura 78.

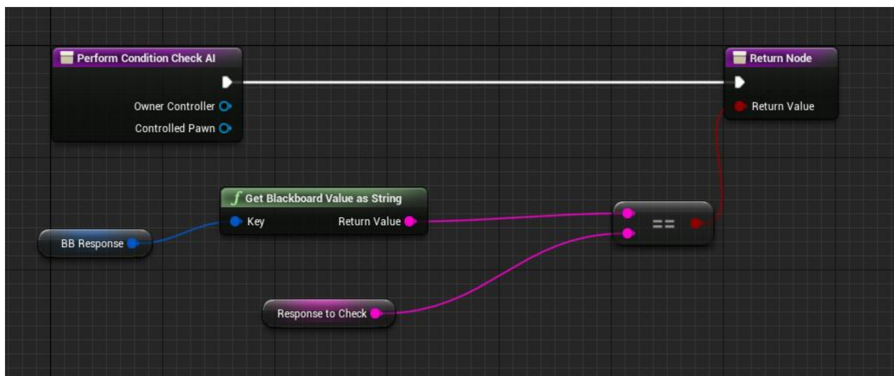


Figura 78: Comprobación entre la respuesta dada y la necesaria para entrar en esa rama.

En una misión de recogida de objetos, el jugador debe recolectar lo necesario y después volver con el NPC que realizó la petición. Para comprobar si la misión está completada hemos creado el nodo D_CheckItemForQuestComplete. Este nodo comprueba si el objeto esta en el inventario del jugador y si es así, lo elimina del mismo y da por completada la misión. Ver Figuras 79 y 80.

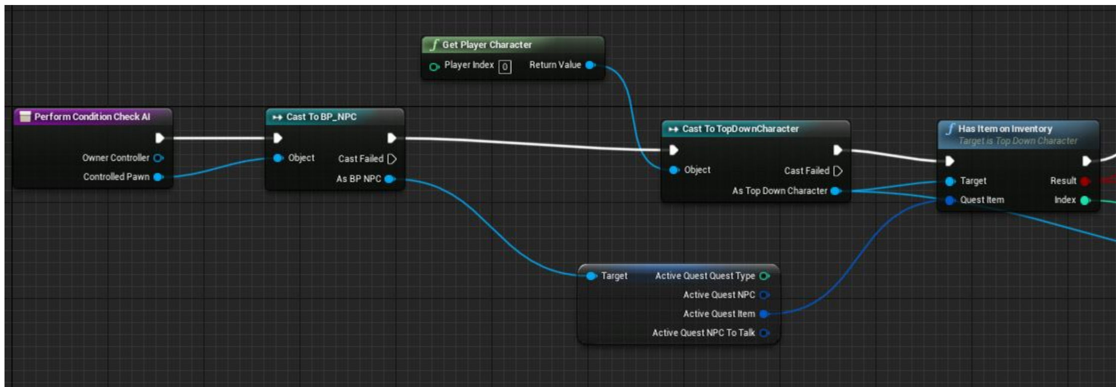


Figura 79: Comprobación si un objeto está en el inventario.

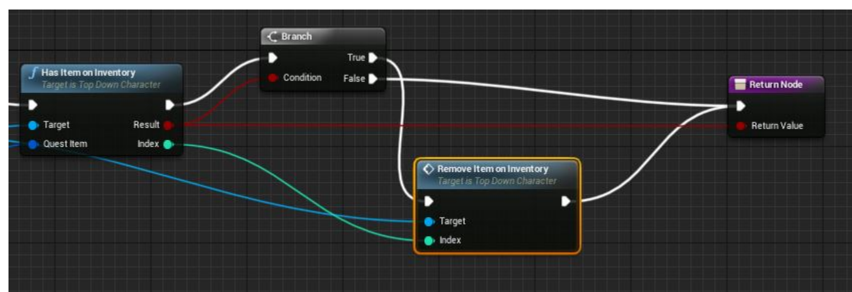


Figura 80: Eliminación de un objeto del inventario.

Para comprobar si el jugador tiene el objeto en su inventario, recorremos todos los objetos y si el nombre de alguno coincide con el que buscamos, es que el objeto que buscábamos está en el inventario. Ver Figura 81.

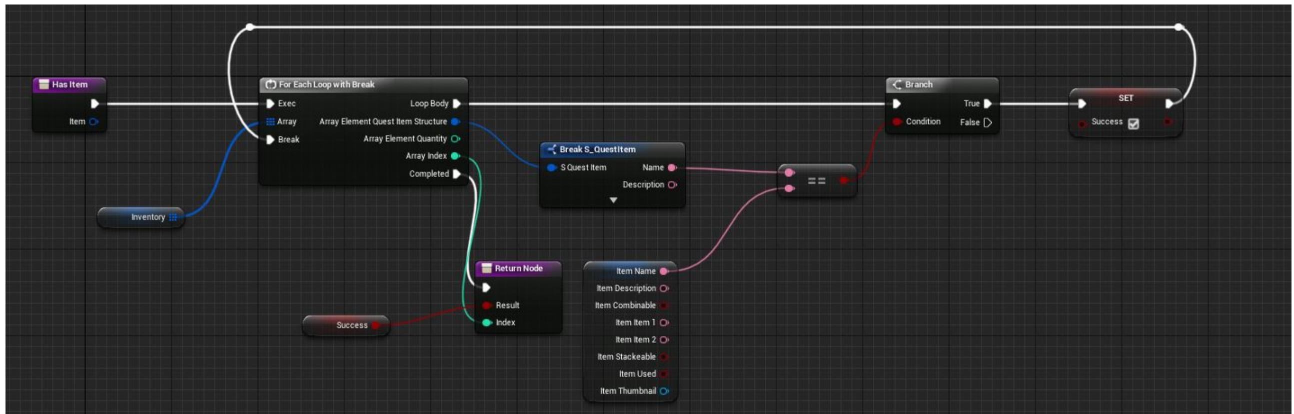


Figura 81: Búsqueda de un objeto en el inventario.

Para eliminarlo tan solo tenemos que usar el índice proporcionado y quitarlo de la estructura de datos del inventario. Ver Figura 82.

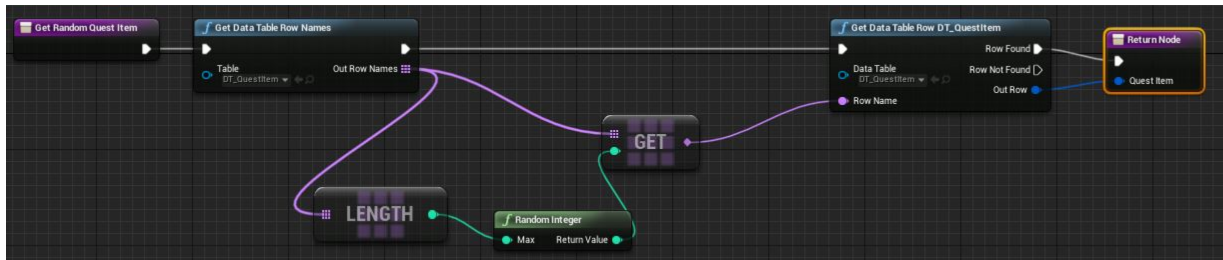


Figura 82: Eliminación de un objeto en el inventario.

En una misión de recogida hemos de comprobar si el NPC está esperando un objeto para escoger una rama dentro del árbol. El nodo D_WaitingObject comprueba la variable del Blackboard para ver si es cierto o no que espera un objeto. Ver Figura 83.

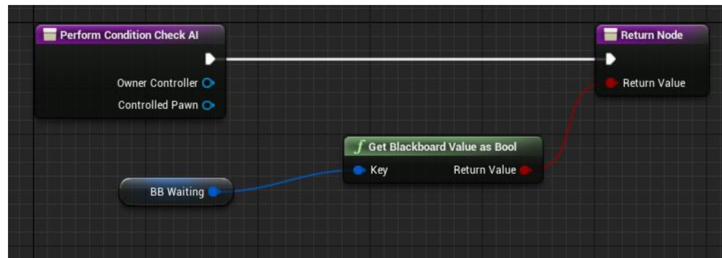


Figura 83: Comprobación si un NPC está a la espera de un objeto.

6.3.5. Behaviour Trees construidos

Para el diálogo asociado a una misión de recogida de objetos hemos construido un árbol (ver Figura 84). En él se ejecutan todas las posibles opciones que explicaremos según veamos las distintas ramas del árbol.

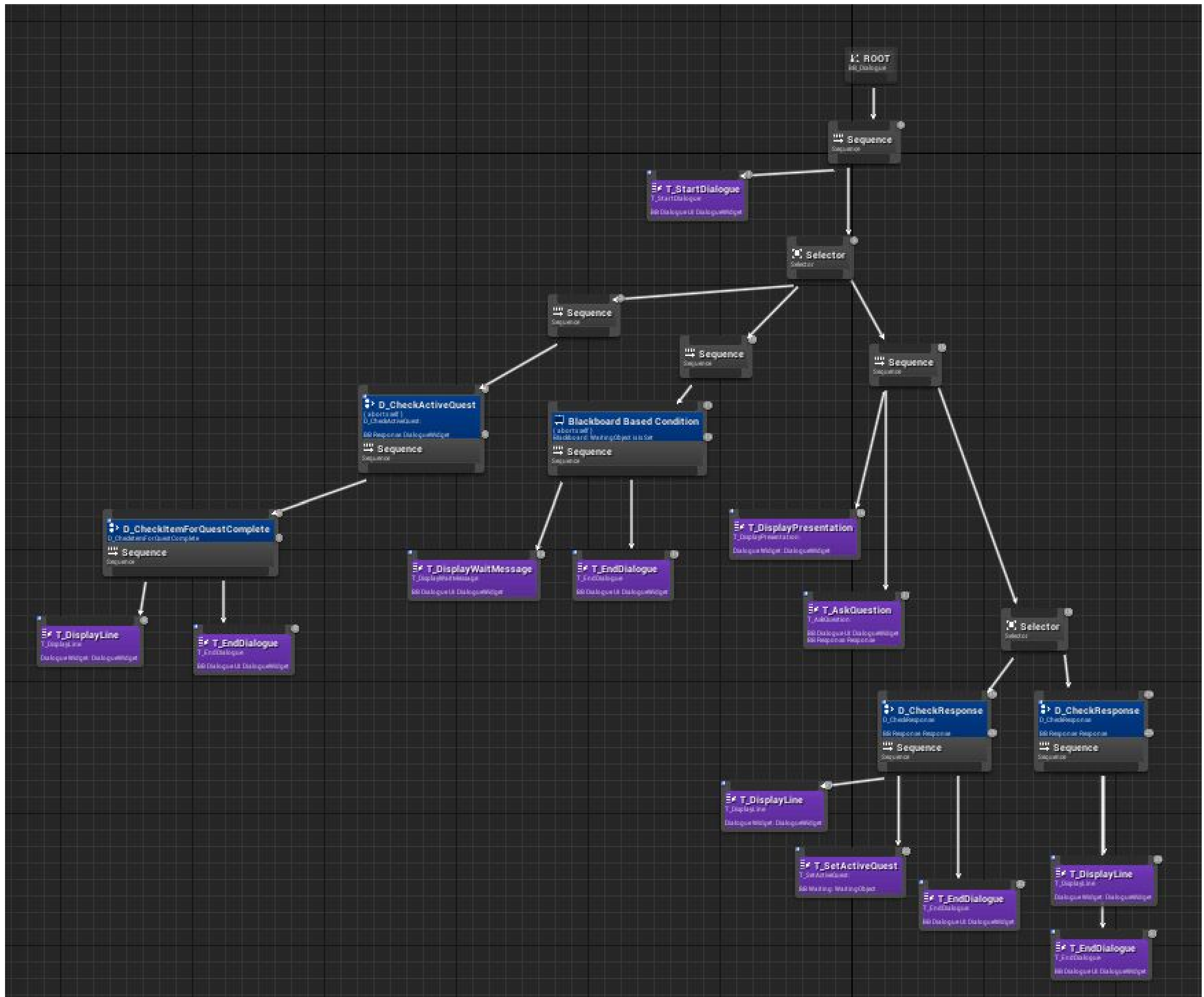


Figura 84: Visión general del árbol de diálogo para una misión de recogida de objetos. En las Figuras posteriores lo explicaremos paso a paso.

Lo primero para empezar un diálogo es iniciarlo con el nodo T_StartDialogue. Ver Figura 85.

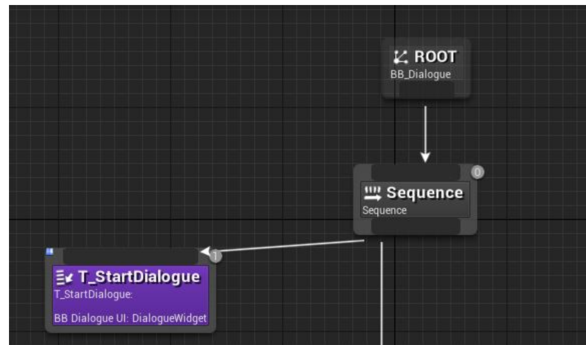


Figura 85: Inicio del Behaviour tree

A continuación, tenemos tres ramas distintas:

- Primera rama: en ella comprobamos primero si el NPC tiene una misión activa. Si es así entonces, comprobamos si el jugador tiene en el inventario el objeto necesario para completar la misión. Si lo tiene, termina el diálogo y se completa la misión, si no lo tiene, se aborta la ejecución de la rama y pasa a la siguiente. Ver Figura 86.

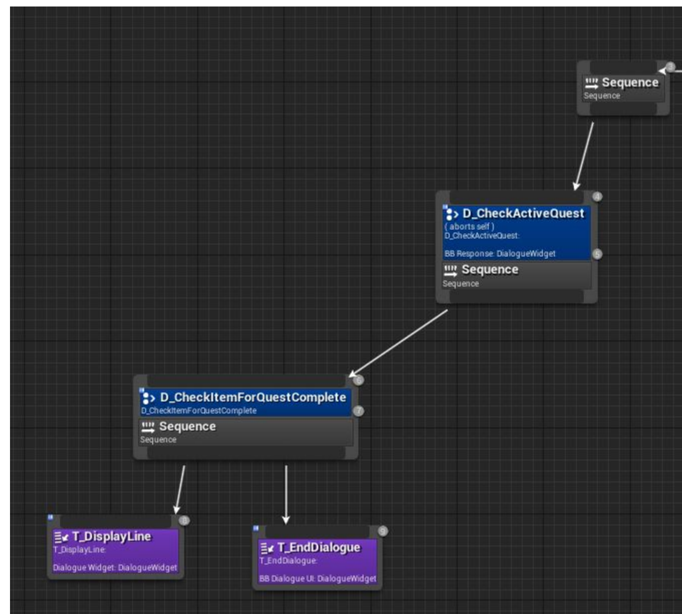


Figura 86: Primera rama.

- Segunda rama: revisamos la variable, que nos indica si el NPC espera un objeto, en el Blackboard. Si lo está esperando, mostramos el mensaje de espera y termina el diálogo. Si no, se aborta la ejecución y pasa a la siguiente rama. Ver Figura 87.

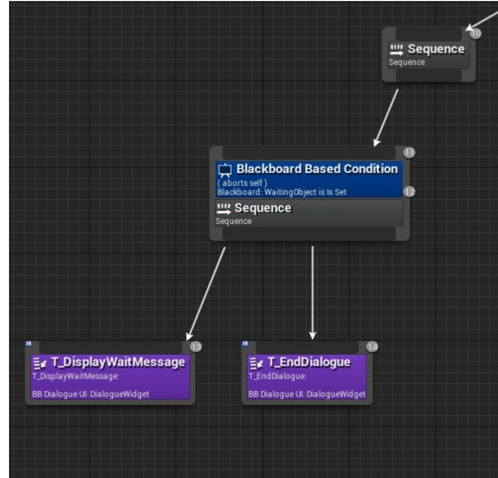


Figura 87: Segunda rama.

- Tercera rama: tras las comprobaciones anteriores, esta rama se ejecuta cuando el NPC no tiene la misión activa y, por tanto, puede activar una misión para el jugador. Lo primero es mostrar la presentación del NPC al jugador para después preguntarle si acepta la misión que se le plantea. Cuando el jugador contesta se comprueba cual ha sido la respuesta. Si ha sido afirmativa se muestra una línea de confirmación, se activa la misión y se acaba el diálogo. En el caso contrario, cuando el jugador no acepta la misión, se muestra un mensaje de confirmación que ha rechazado la misión y se termina el diálogo. Ver Figura 88.

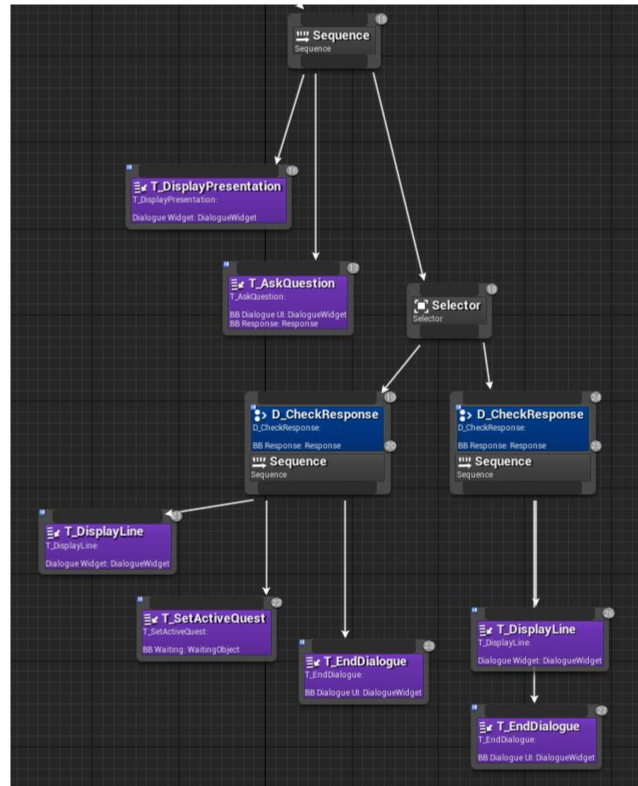


Figura 88: Tercera rama.

Para el diálogo asociado al inicio de una misión de diálogo, hemos creado otro Behaviour Tree. Es prácticamente igual que la tercer rama (ver Figura 88) del árbol anterior, con la diferencia que hemos de iniciar el modo de diálogo, y el nodo para activar la misión es distinto. Ver Figura 89.

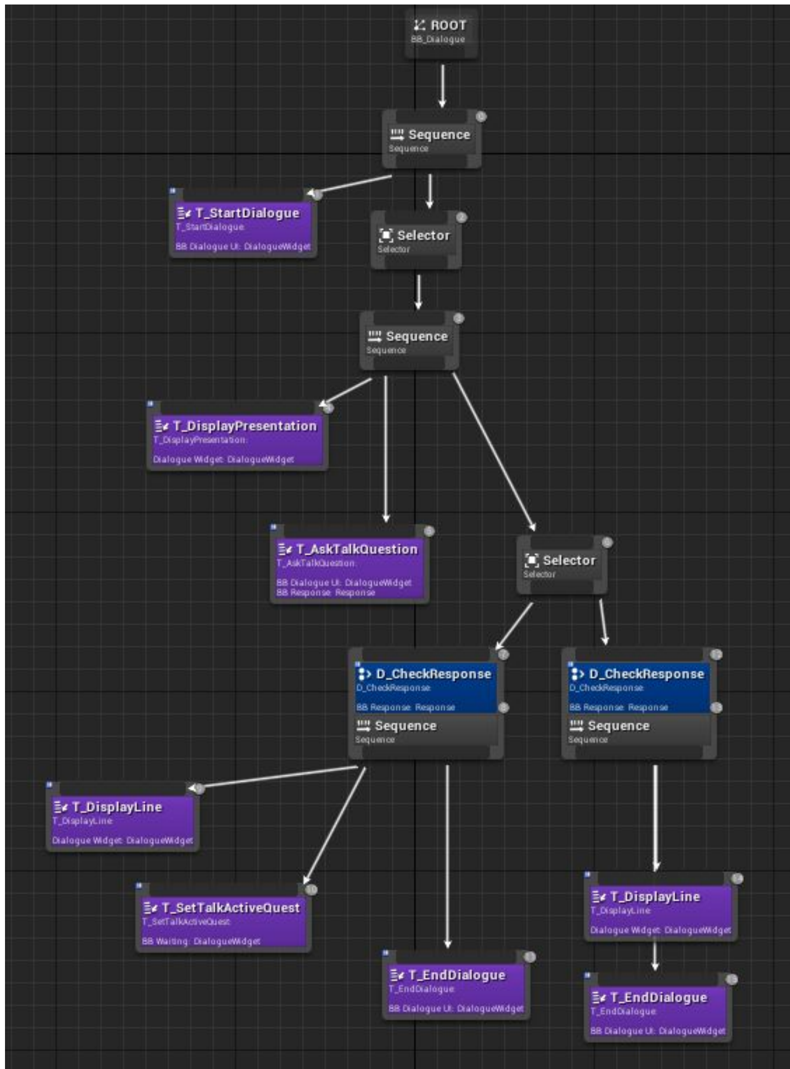


Figura 89: Visión general del árbol para iniciar una misión de diálogo.

Para finalizar esta misión, tenemos un árbol que comprueba si el NPC tiene una misión activa. Si es así, inicia el diálogo, muestra el texto correspondiente y termina el diálogo. Ver Figura 90.

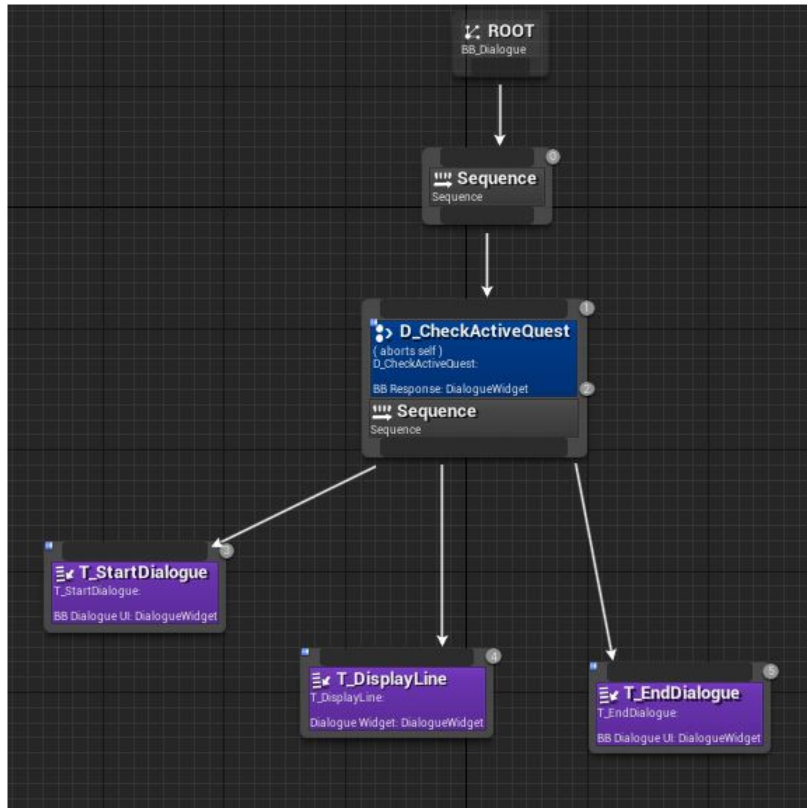


Figura 90: Visión general del árbol para finalizar una misión de diálogo.

6.4. Inventario

Para que el jugador o cualquier NPC pueda tener objetos hemos creado un componente específico para ello. Lo interesante de que sea un componente es que se puede añadir a cualquier actor, como al jugador, los NPC o incluso otros objetos, lo que proporciona mucha flexibilidad a la hora de diseñar.

El inventario es una estructura que contiene otras más pequeñas compuestas por un objeto y la cantidad que tenemos del mismo. Estas estructuras dentro de otra estructura las hemos llamado “slot“. Ver Figura 91.

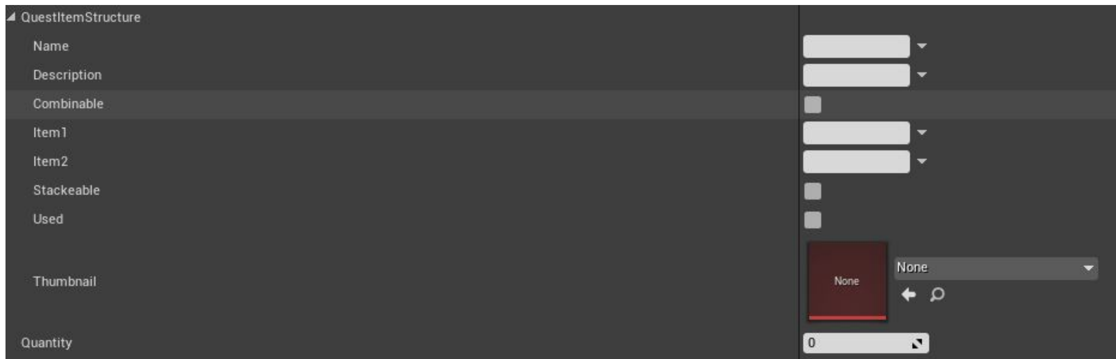


Figura 91: Atributos de un “slot“.

6.4.1. Preparación

Una vez añadido el componente del inventario a cualquier actor, debemos prepararlo. Para ello lo primero es fijar el número de “slots“ que tendrá el inventario. Ver Figuras 92 y 93.

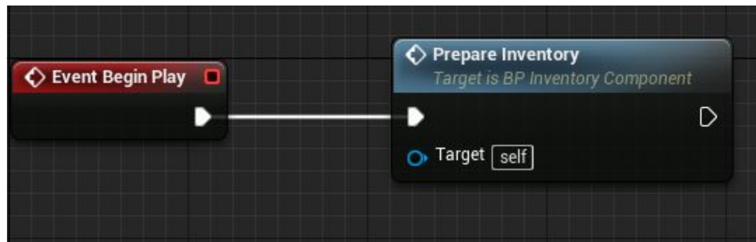


Figura 92: Inicio de un componente de inventario.

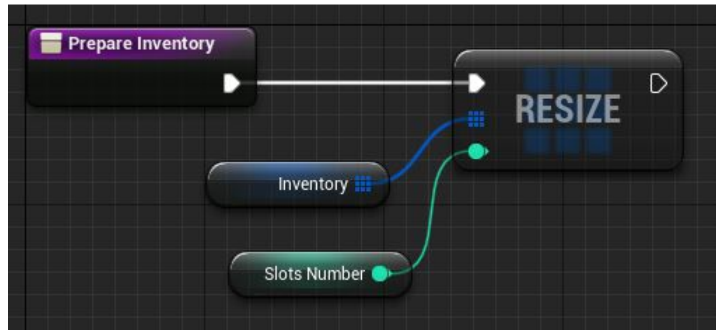


Figura 93: Preparación de un componente de inventario.

6.4.2. Añadir objetos

Para añadir un objeto al inventario usamos el método "Add To Inventory", el cual recibe el "slot" a añadir. Lo primero que se debe comprobar es si el objeto es apilable o no. Si lo es, tendremos que buscar si ya existe una pila de ese objeto creada para añadirlo a ella. Si no lo es, crearemos una nueva pila para ese objeto. Ver Figura 94.

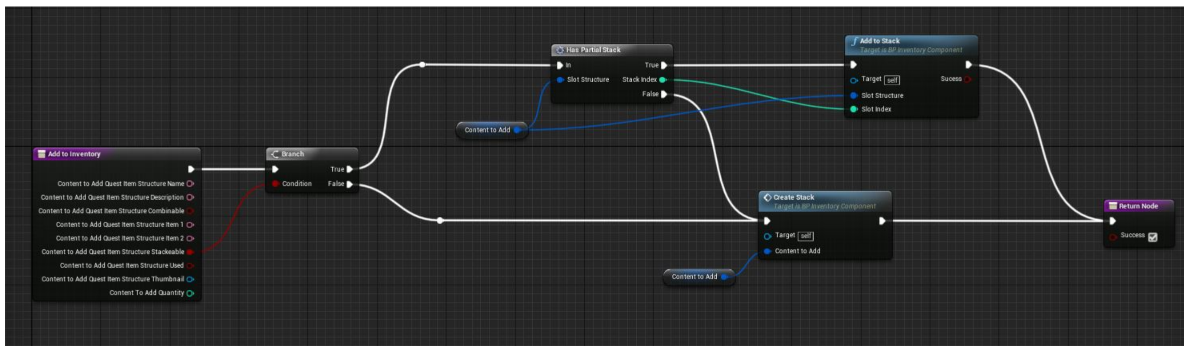


Figura 94: Añadir objeto al inventario.

Para comprobar si ya existe una pila creada de un objeto usamos la macro "Has Partial Stack". Esta macro recorre los objetos del inventario, comparándolos con el que recibe. Si coinciden, para de recorrer y devuelve el índice donde encontró la coincidencia. Ver Figura 95.

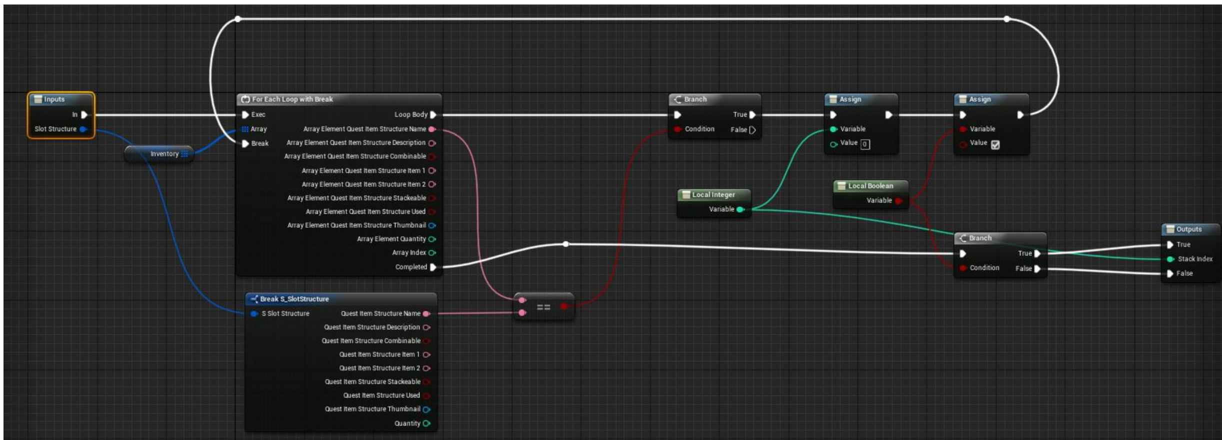


Figura 95: Búsqueda de una pila de objetos en el inventario.

Si debemos crear una nueva pila usamos el método “Create Stack“. Para crear la pila buscamos un “slot“ vacío (cuya cantidad sea 0) y añadimos en ese hueco nuestro objeto. Ver Figura 96.

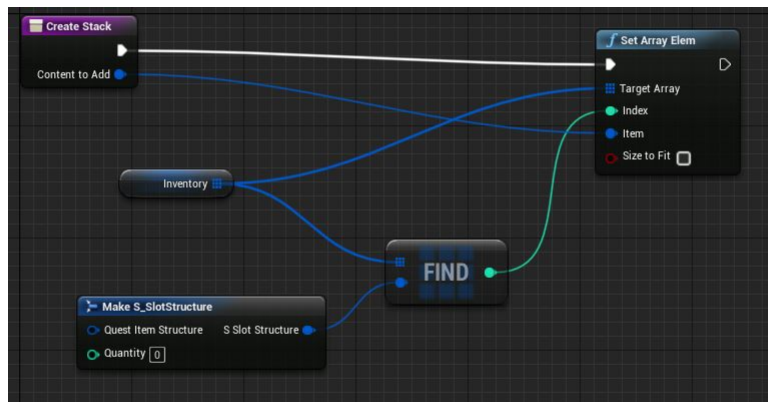


Figura 96: Crear una pila de objetos.

En el caso que exista una pila usamos el método “Add To Stack“. Este método recibe el índice donde está situada la pila dentro del inventario. Para añadir el objeto obtenemos la cantidad actual y la cantidad que debemos añadir. Sumando ambas cantidades obtenemos la cantidad final que pondremos en el “slot“ del inventario. Ver Figura 97.

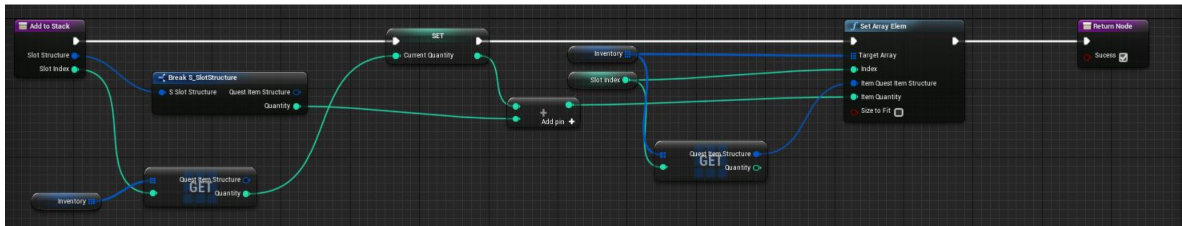


Figura 97: Añadiendo a una pila de objetos.

6.4.3. Existencia de un objeto en el inventario

Algo muy común es comprobar si existe un objeto en el inventario, para ello usamos el método “Has Item“. Este método recorre los objetos del inventario comparándolos con el que recibe. Si coinciden, para de recorrer y devuelve el índice donde encontró la coincidencia, junto con un booleano que indica si ha sido encontrado o no. Ver Figura 98.

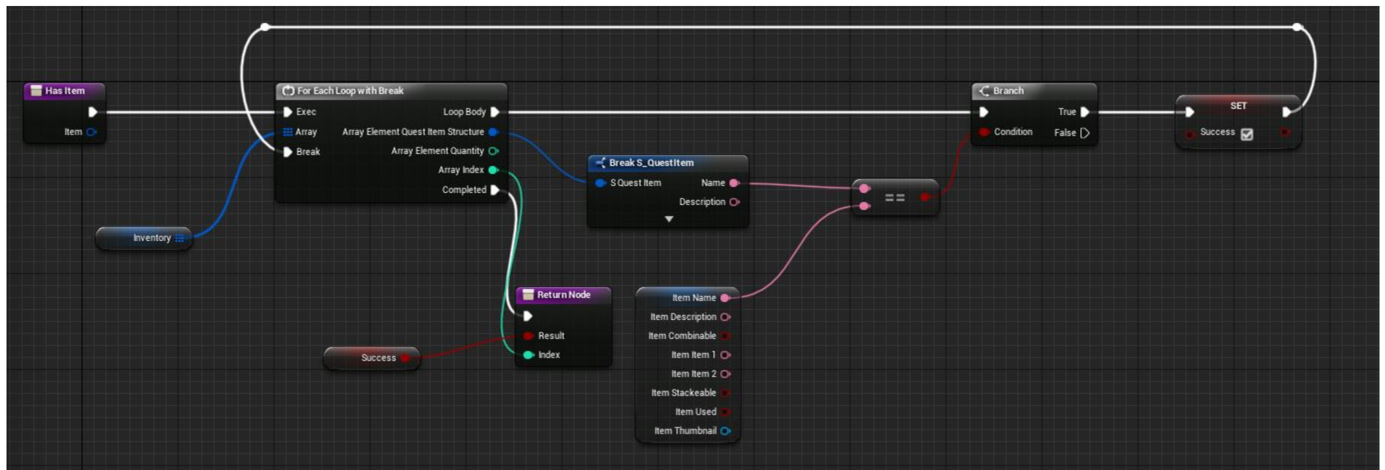


Figura 98: Existe el objeto en el inventario.

6.4.4. Eliminar un objeto del inventario

Para eliminar un objeto del inventario usamos el método “Remove Item“, el cual recibe el índice del objeto a eliminar dentro del inventario. Lo ideal es combinar el método “Has Item“, que proporciona el índice del objeto si existe en el inventario, con “Remove Item“ que elimina el objeto del índice indicado. Ver Figura 99.

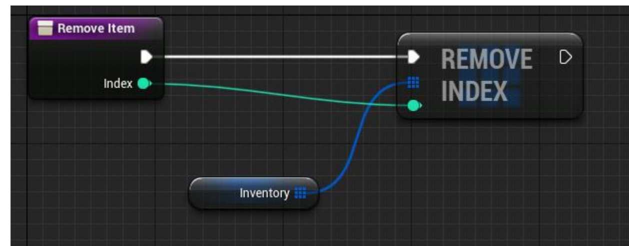


Figura 99: Eliminar un objeto del inventario.

6.4.5. Mostrar u ocultar el inventario

Para mostrar u ocultar el contenido del inventario usaremos el método “Toggle Inventory“. Lo primero que debemos comprobar es si lo vamos a ocultar o a mostrar en pantalla. Para ello, veremos si el objeto que muestra el inventario existe ya en la interfaz o no. Ver Figura 100.

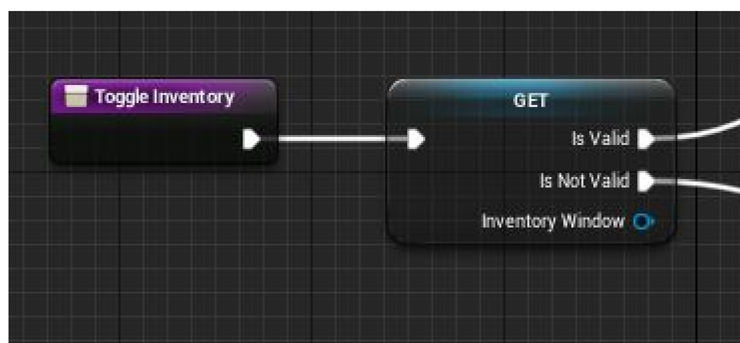


Figura 100: Eliminar un objeto del inventario.

En caso de que exista la ventana, debemos eliminarla y cambiar el valor de la variable. Ver Figura 101.

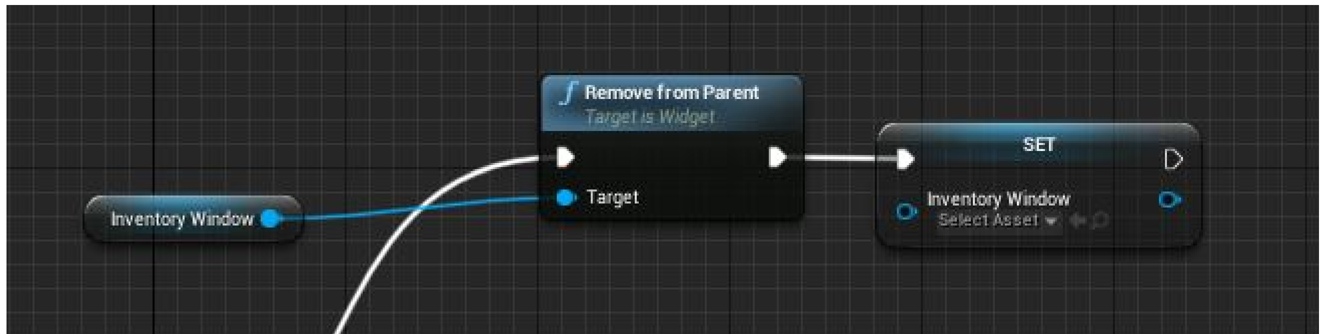


Figura 101: Comprobar si existe la ventana de inventario.

En caso de que no exista la ventana, debemos crearla y añadirla a la pantalla. Ver Figura 102.

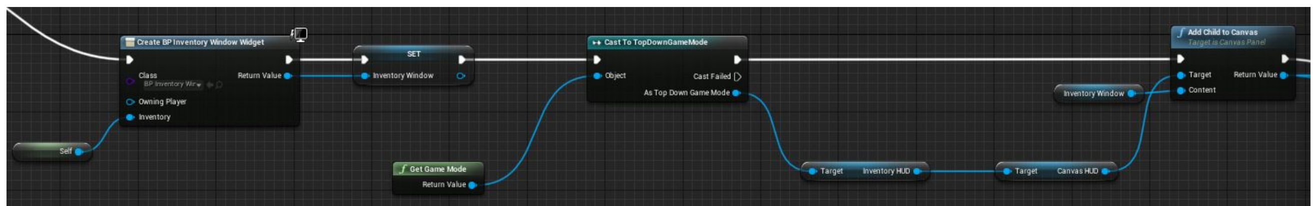


Figura 102: Creando la ventana de inventario.

Una vez tenemos creada la ventana, definimos su tamaño, alineación, a que zona de la pantalla anclarla (así siempre se mostrará en esa zona) y la posición en referencia a la zona donde estará anclada. Ver Figura 103.

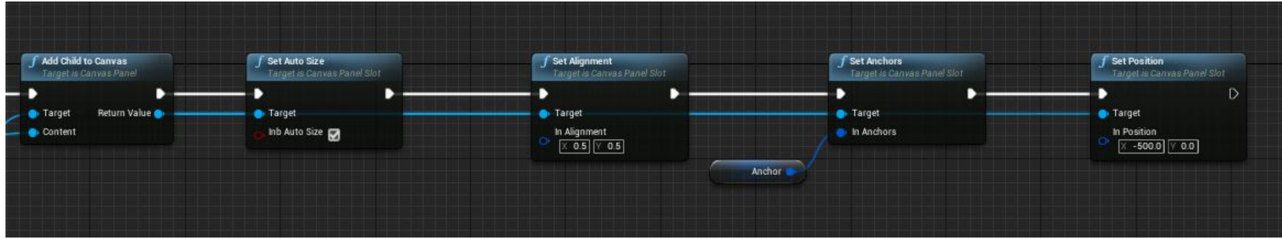


Figura 103: Posicionamiento de la ventana de inventario.

6.5. Movimiento y menú de interacción

Unreal Engine tiene muchos modos de movimiento ya implementados en sus plantillas. Hemos utilizado una de ellas, por lo que el movimiento ya está implementado, pero hemos añadido modificaciones para que se adapte a nuestras necesidades.

Para moverse se utiliza el ratón, haciendo clic izquierdo en una localización nos movemos a ella. El método que mueve el personaje controlado por el jugador es “Move to Hit Location“. Este método recibe un “Hit“ provocado por el clic del ratón y del cual obtenemos la posición en el escenario a la que se moverá el personaje. Si la localización del clic ha sido demasiado cercana al personaje, el personaje no se moverá. Ver Figura 104.

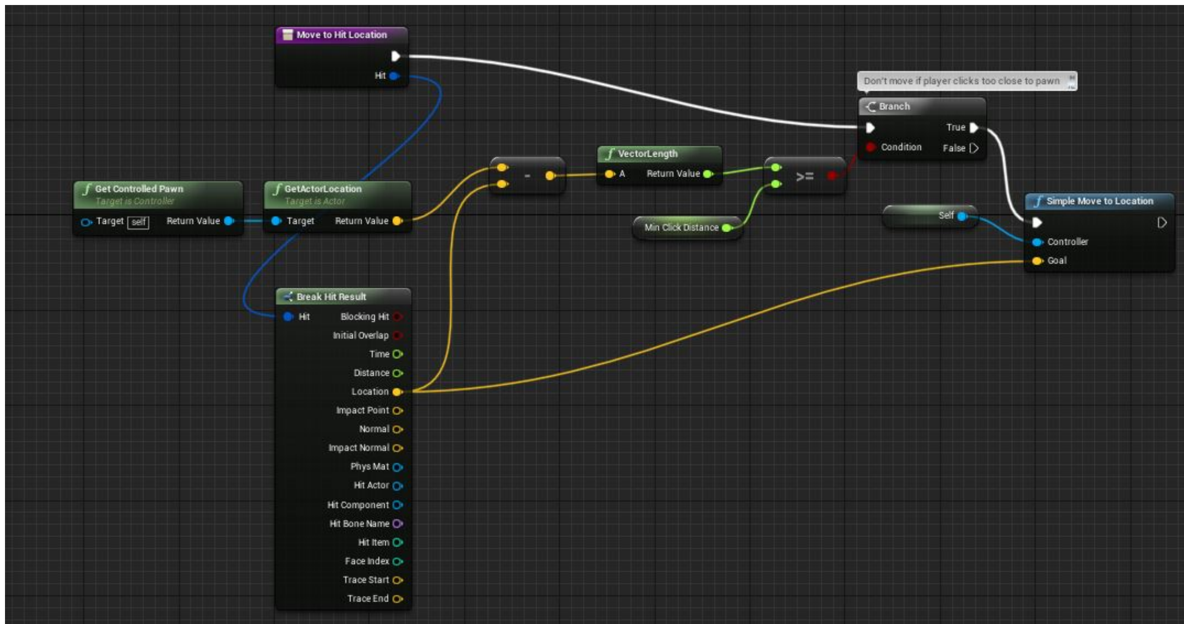


Figura 104: Movimiento del personaje.

Cuando el jugador interactúa con un NPC o un objeto, no puede moverse mientras el menú de interacción esté en pantalla. Para lograr esto hemos modificado el movimiento incluyendo un booleano que impide que se ejecute el método "Move to Hit Location" si el movimiento no está permitido. Ver Figura 105.

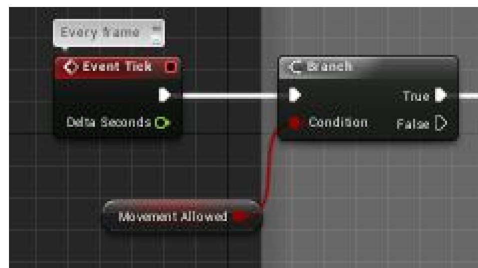


Figura 105: Limitación del movimiento del personaje.

El valor de esta variable cambia dependiendo si el menú está en pantalla o no. Para activar el modo menú usamos el método "Menu mode". Este método cambiar el valor de la variable anterior para evitar que el jugador se muestre, asigna las variables para que sepamos

cual es el actor o el objeto con el que interactúa, y por último muestra la interfaz del menú. Ver Figura 106.

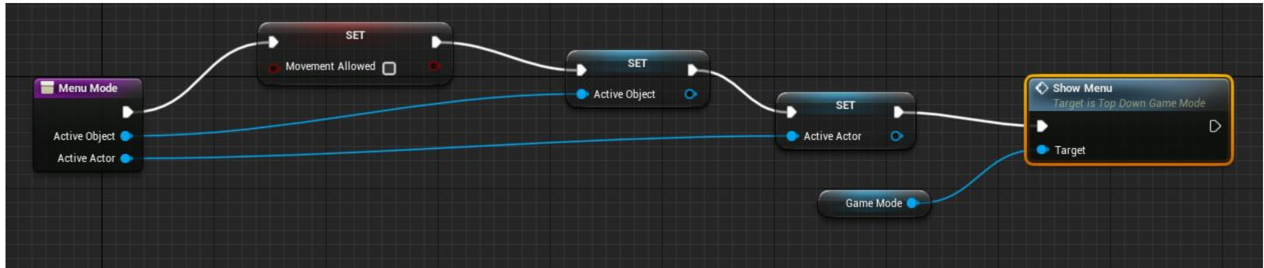


Figura 106: Activación del modo menú.

Para desactivar el modo menú usamos el método "Disable Menu mode". Este método es el encargado de volver a permitir el movimiento y quitar el menú de la interfaz. Ver Figura 107.

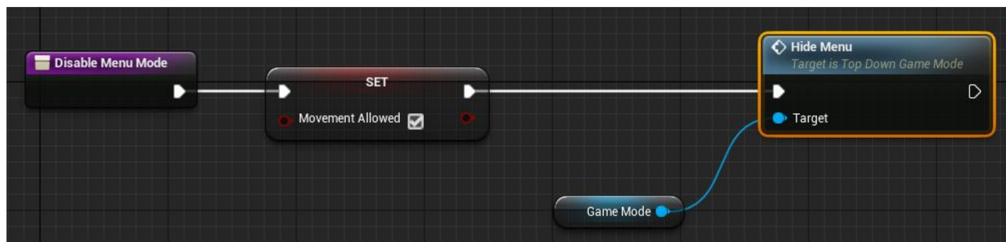


Figura 107: Desactivación del modo menú.

6.6. Objetos interactuables

Por el escenario están repartidos los objetos con los que el jugador puede interactuar. Estos objetos son los que hemos nombrado como "Interactuable Object". Estos objetos pueden ser recogidos, inspeccionados e incluso intentar hablar con ellos.

6.6.1. Recoger un objeto

Los objetos que se pueden recolectar a través del menú de interactuar van al inventario. Cuando un objeto se recoge, usamos el método "Collect". Este método añade el objeto al inventario, desactiva el modo menú y, si pudo añadir el objeto al inventario, lo elimina del escenario. Ver Figura 108.

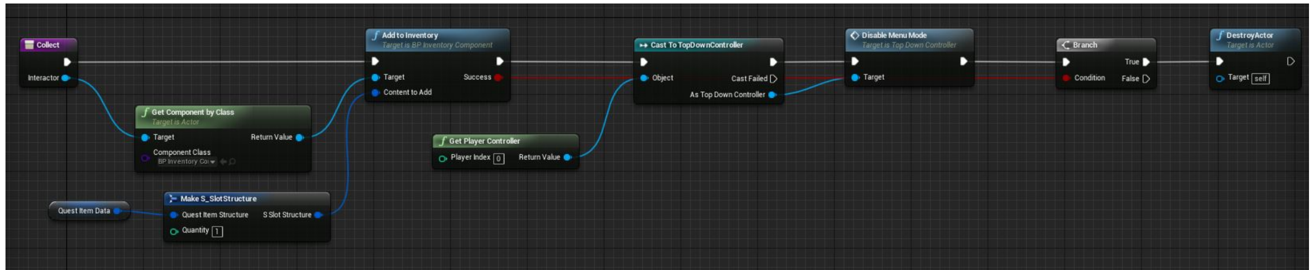


Figura 108: Método para recoger un objeto.

6.6.2. Inspeccionar un objeto

El jugador puede inspeccionar los objetos para poder obtener información adicional acerca de ellos. Para mostrar el texto utilizaremos un método similar al ya explicado anteriormente en los Behaviour Trees. Mostramos la interfaz de diálogo, inicializando el nombre y el texto que pondremos en ella, ocultamos el menú y por último, activamos el evento tick del objeto. Ver Figura 109.

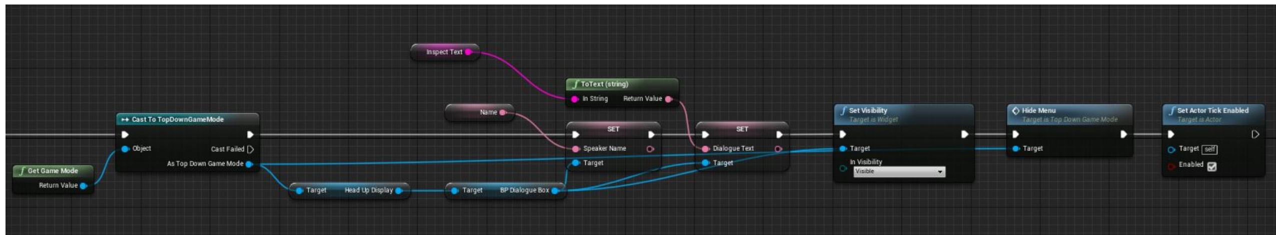


Figura 109: Inspeccionar objeto.

6.6.3. Evento tick

El evento tick se ejecuta cada cierto tiempo cuando esta activado. En este caso lo usamos para comprobar si el jugador pulsa una tecla concreta (botón derecho del ratón). Si lo hace, terminamos el diálogo. Ver Figura 110.

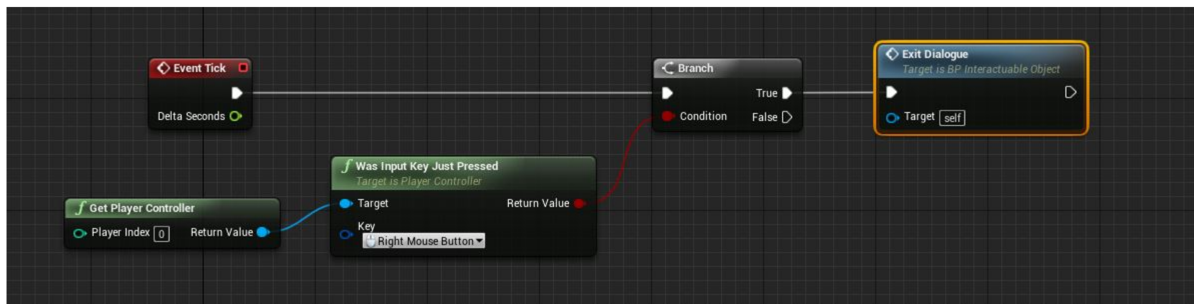


Figura 110: Evento tick.

6.6.4. Finalizar el diálogo

Para finalizar el diálogo haremos lo mismo que en los Behaviour Trees. Obtenemos acceso a la interfaz y la ocultamos, permitimos de nuevo el movimiento y, por último, desactivamos el evento tick. Ver Figura 111.

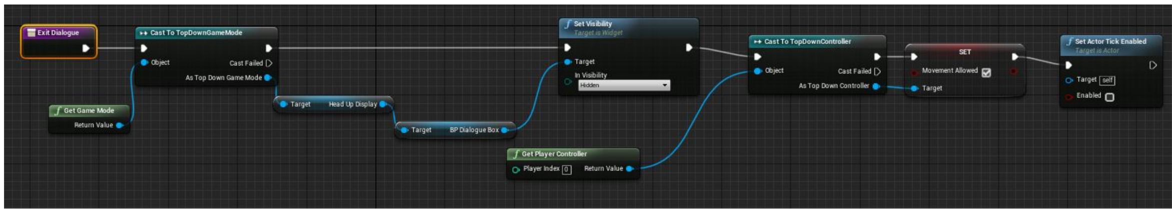


Figura 111: Exit dialogue.

6.6.5. Hablar con un objeto

El jugador puede tratar de hablar con los objetos, estos le responderán o no, dependiendo del objeto. Para mostrar el texto utilizaremos un método similar al ya explicado anteriormente en los Behaviour Trees. Mostramos la interfaz de diálogo, inicializando el nombre y el texto que pondremos en ella, ocultamos el menú y, por último, activamos el evento tick del objeto. Ver Figura 112.

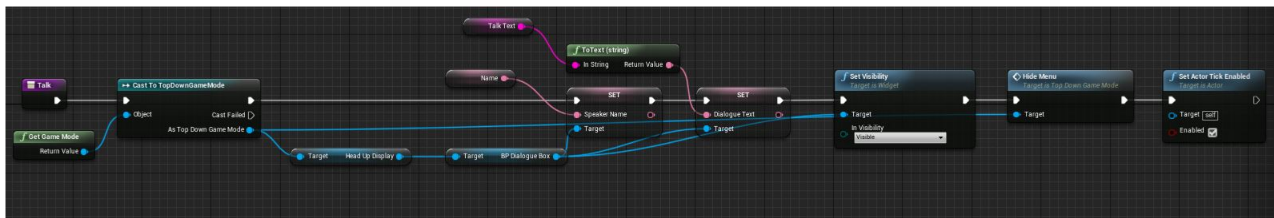


Figura 112: Hablar con un objeto.

7. Resultados

7.1. En función de los objetivos

El objetivo de este proyecto es crear una aventura gráfica con puzzles generados proceduralmente. Este objetivo se ha cumplido en gran medida ya que hemos desarrollado un laberinto que se genera de manera procedural y una historia que se puede ramificar.

El laberinto creado varía en función de una semilla y tras las pruebas en tiempo de ejecución se observa que, con un pequeño coste en recursos, se obtienen grandes cambios. Variando el valor de la semilla conseguimos cambiar entre los laberintos de la Figura 113 al laberinto de la Figura 114.

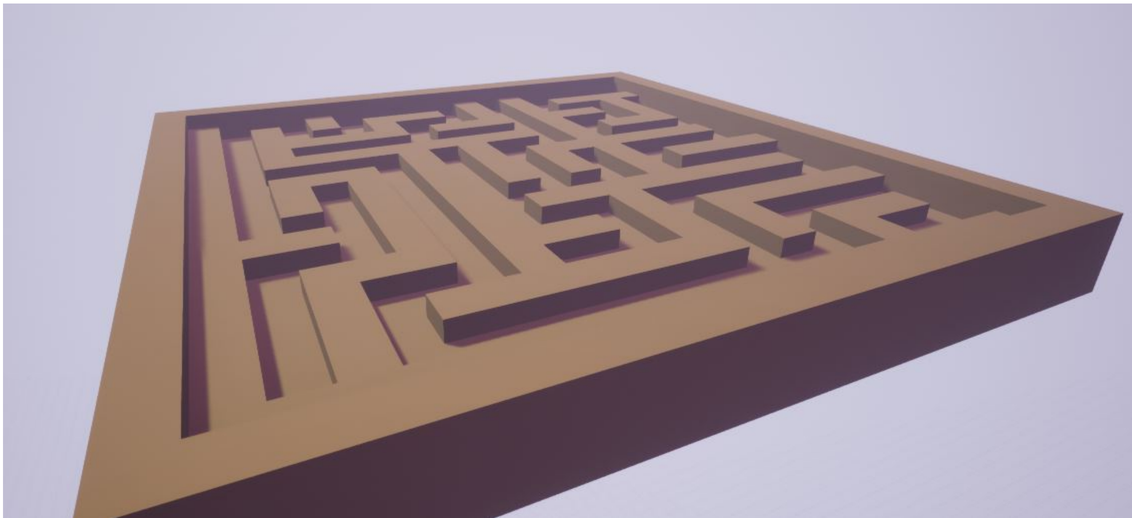


Figura 113: Primer laberinto generado.

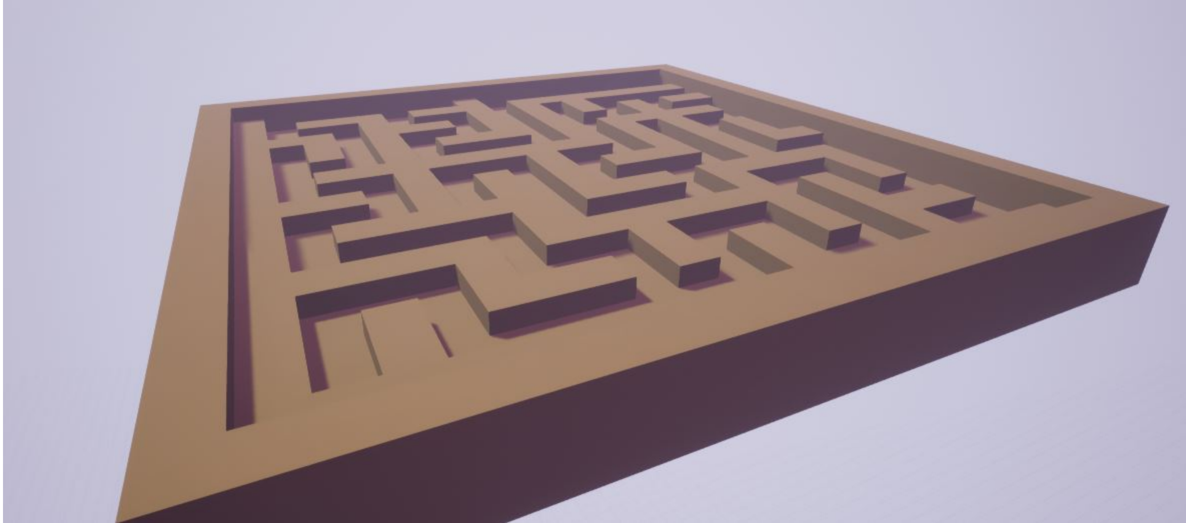


Figura 114: Segundo laberinto generado.

Este puzzle es un claro ejemplo de elemento reutilizable: con tan solo un código, podemos tener una combinación prácticamente infinita para que el jugador no sea capaz de solventarlo de la misma forma que lo hubiera hecho otras veces.

El sistema de misiones que hemos creado es muy variable, por lo que obtendremos un número de misiones distintas bastante amplios. Si tenemos un número n de NPCs y un número o de objetos podemos llegar a tener una combinación de $n(n \times o)$ misiones de recolección de objetos distintas y para las misiones de dialogo podemos tener $n(n-1)$ misiones distintas. Ver Figuras 115, 116, 117, 118 y 119.

Figura 115: Inicio de una misión de dialogo.

Figura 116: Final de una misión de dialogo.

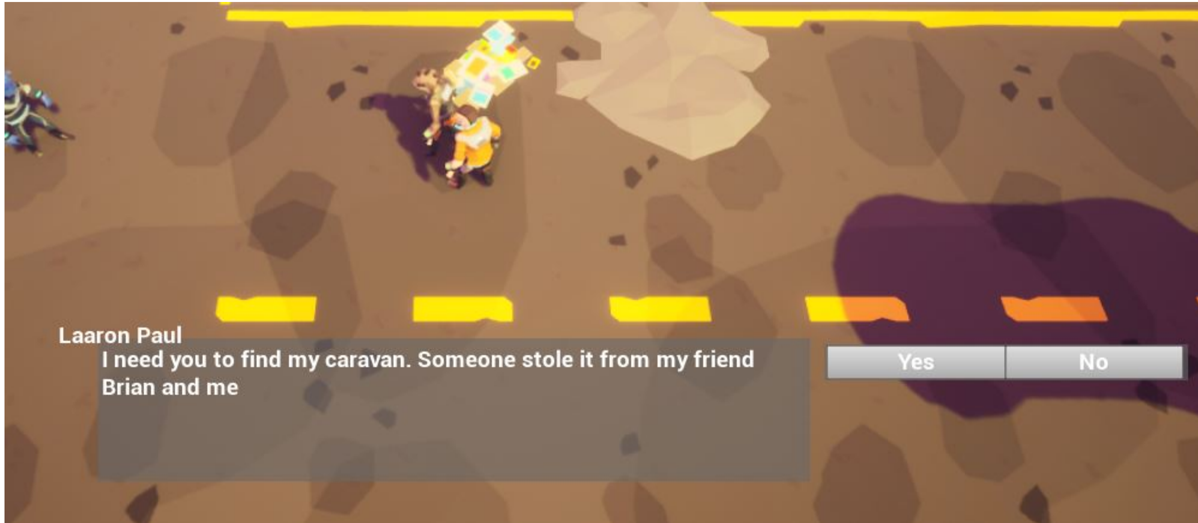


Figura 117: Inicio de una misión de recogida de objetos.

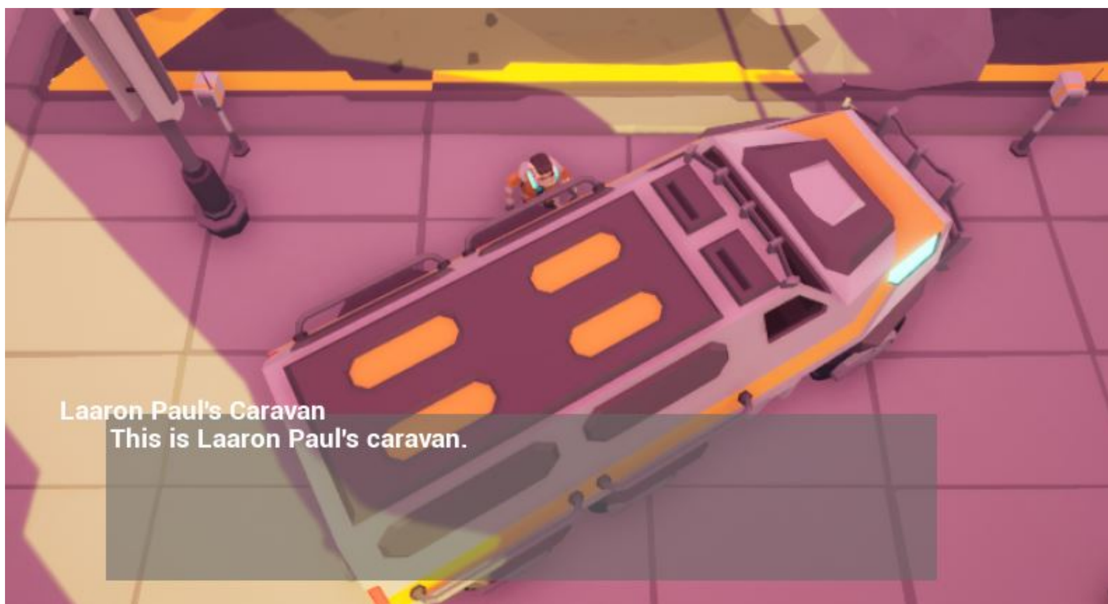


Figura 118: Encontrado el objeto de la misión.

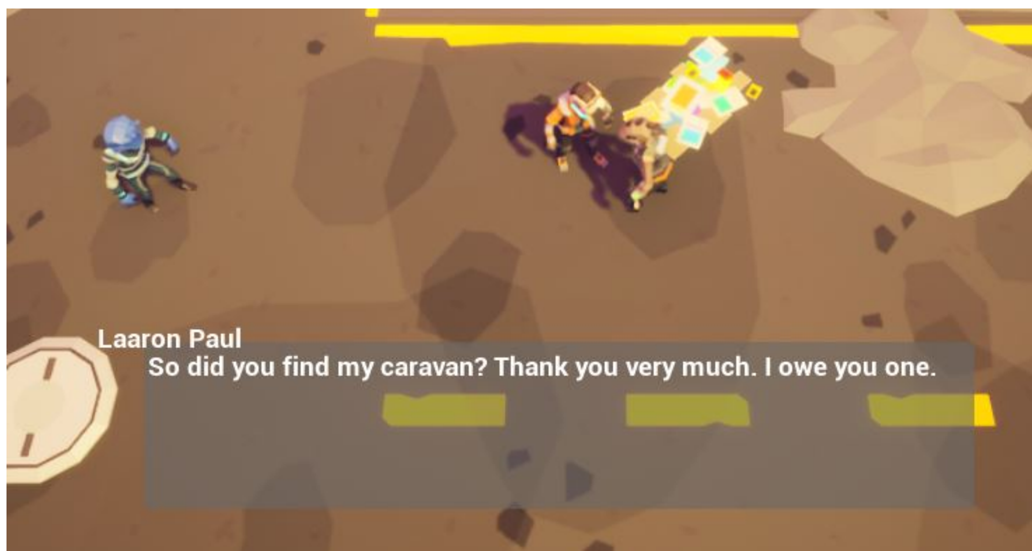


Figura 119: Finalizando la misión.

7.2. Visión legal

En nuestro proyecto no se recogen datos de ningún tipo ni se almacenan por lo tanto no tiene ninguna implicación legal en cuanto a la LOPD (Ley Orgánica de Protección de Datos). Tampoco incluye ningún servicio de pago por lo que no tiene ninguna implicación con LSSICE (Ley de Servicios de la Sociedad de la Información).

8. Conclusiones

En relación a los objetivos, se han conseguido todos los propuestos, y como consecuencia de una programación reutilizable, hemos obtenido un sistema que permite ser utilizado para más juegos de una manera simple. Por ejemplo, los juegos creados por el estudio de Telltale games son aventuras gráficas que comparten las mismas mecánicas pero cambian los personajes y la historia de cada uno. Puliendo algunos aspectos en cuanto a la reutilización podríamos llegar a conseguir el mismo sistema que tienen en Telltale Games, lo que proporcionaría un abaratamiento enorme de costes en los futuros desarrollos.

Teniendo en cuenta que en el mundo de los videojuegos la generación procedural no es un ámbito muy habitual y, por tanto, no hay demasiada información sobre su implementación en motores, consideramos que el resultado obtenido ha sido bueno, incluso superando las expectativas iniciales.

Por supuesto, existen cosas que mejorar pero habiendo superado los objetivos iniciales e incluso consiguiendo cosas inesperadas, concluimos que el proyecto se ha desarrollado con éxit,o pese a tener que modificar la planificación inicial por el COVID-19.

9. Trabajo futuro

Para el trabajo futuro tenemos varias tareas pendientes:

- **Reutilización**: Hay algunos aspectos que no son del todo reutilizables y fácilmente modificables, por lo que deberíamos trabajar para mejorar esto de cara a conseguir un sistema totalmente reutilizable. Algunos textos están directamente en el código, sería mucho mejor poder obtenerlos de un archivo, o que se pueden introducir en la interfaz de Unreal Engine cuando se construye la escena.

- **Variedad en los puzzles**: de momento tan solo contamos con el laberinto generado de forma procedural, la idea es conseguir más tipos de esta manera para tener variedad dentro del sistema. Los puzzles tipo Sokoban son bastante divertidos y pueden ser fáciles de generar proceduralmente con algoritmos similares a los que hemos utilizado, pero adaptándolos a las particularidades del puzzle.

- **Interfaz**: la interfaz ahora mismo es muy pobre, está hecha tan solo con fines de probar las mecánicas. Sería conveniente diseñar una interfaz con una estética futurista basada en colores azules y toques de neón para que se adapte a la estética general. En los diálogos tan solo aparece el nombre del NPC que habla, se mejoraría poniendo una imagen del personaje también.

- **Mecánicas**: una nueva mecánica que podríamos añadir son opciones de dialogo extras con un porcentaje de éxito y fracaso. Por ejemplo, una opción que si tiene éxito descubre un hecho relevante de la trama y si fracasa se cierran las líneas de conversación con ese NPC. También el hecho de tener un objeto podría abrir nuevas opciones de dialogo con algún NPC.

Referencias

[1] "Gamificación: ¿cómo saber los tipos de jugadores en nuestra oficina?." https://www.laescuelita.cr/post/gamificaci%C3%B3n-tipos-de-jugadores. (Accessed on 08/10/2020).

[2] N. A. Barriga, "A short introduction to procedural content generation algorithms for videogames," *International Journal on Artificial Intelligence Tools*, vol. 28, no. 02, p. 1930001, 2019.

[3] "Los juegos para móviles generan mucho más dinero que los juegos de pc." https://hardzone.es/2019/03/18/juegos-moviles-mas-dinero-pc/. (Accessed on 12/11/2019).

[4] "Fortnite daily mobile revenue reaches $2m — gamesindustry.biz." https://www.gamesindustry.biz/articles/2018-07-24-fortnite-daily-mobile-revenue-reaches-usd2m. (Accessed on 12/11/2019).

[5] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[6] B. De Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, 2019.

[7] "Polygon - sci-fi city pack — synty store." https://syntystore.com/collections/frontpage/products/polygon-sci-fi-city. (Accessed on 08/24/2020).