Bakumapu

Diseño técnico

Estudio %s - 2022-3-10_v0.0.27

Índice

1. Introducción

Este documento busca ser una herramienta de diseño que guíe tanto el flujo de trabajo del desarrollo como la implementación del código en el motor de juegos **Godot**, además de ser una referencia de consulta para detalles concretos y obtener una visión general del funcionamiento interno del juego.

Para facilitar el acceso a la información se agrupan los contenidos en secciones afines y se añade un completo índice de contenidos junto a referencias cruzadas a las distintas secciones relevantes dentro del mismo texto.

En concreto, el documento incluye información con respecto al flujo de trabajo (Metodología Kanban, **??**), a la propuesta de diseño o arquitectura (Modelado del software), al uso de sistemas de control GIT (**??**), a consideraciones de Internacionalización y a muchos otros detalles relevantes para un software de esta magnitud y complejidad.

1.1. Cómo usar este documento

En primer lugar, lo ideal es una lectura completa para tener una noción general del funcionamiento del software y los objetivos del diseño. Luego, la idea es utilizarlo a modo de referencia o mapa de ruta mientras se implementa el código actualizando y editando su contenido continuamente.

El texto estará disponible para su modificación en el formato TEX dentro del repositorio de documentación del proyecto y disponible para su lectura en HTML y en PDF.

El manejo del documento se abordará con más detalle en el apartado El documento de Diseño técnico.

1.2. Información importante.

TL;DR

Los aspectos más relevantes con respecto a la implementación de código son los siguientes:

1. Desarrollo dirigido por tests (TDD) y Refactorización: Rendimiento y limpieza.
2. Metodología Kanban, Repositorio, El documento de Diseño técnico y Formato y documentación dentro del código.
3. Funcionamiento general y Funcionamiento del sistema de managers.
4. Organización del proyecto.

1.3. Lista de requerimientos

A continuación se presenta una lista inicial de requerimientos o consideraciones del programa. Este listado no es fijo y se irá ajustando durante el desarrollo.

1. Enfoque en la facilidad de agregar y editar contenido sin escribir código.
    a) Locaciones.
    b) Quests.
    c) Ítems.
    d) Personajes.
    e) Cutscenes.
    f) GUI.
2. Multiplataforma: Inicialmente PC y Android.
3. Multilenguaje: Español e inglés (en primera instancia).
4. Pixelart.

5. Isométrico 2:1.
6. Relación de aspecto 16:9.
7. Resolución ajustable a dispositivos Android. En PC a 720p y 1080p.
8. Tasa de refresco idealmente mayor a 60 FPS.
9. Ajuste dinámico de la velocidad del juego.
10. Ajuste de stats de ítems.
11. Mixes de audio estéreo:
    a) Master.
    b) Música.
    c) Ambiente.
    d) SFX.
    e) Ataques.
    f) Props.
    g) UI.
12. Seguimiento de las decisiones en las quests (quests path).
13. States de guardado solo en locaciones habilitadas del mapa.
    a) Ítems.
    b) EXP.
    c) HP y mana.
    d) Ubicación.
    e) Active Quests.
    f) Quest path.
14. Tácticas y equipo de la party.
15. IA de tácticas de la party.
16. IA enemiga.

17. Pathfinding de NPC.
18. Sistema de magia y poderes.
19. Sistema de partículas para ataques y locaciones.
20. Sistema de iluminación.
21. Interacción con props.
22. Dificultad ajustable en menú.

1.4. Listado de software y herramientas de producción

1.4.1. Godot v3.x

Se utilizará como herramienta principal de desarrollo el motor de videojuegos multiplataforma open source **Godot** en su rama 3.x estable (3.4 actualmente). Ante el lanzamiento de la versión 4.0 se estudiará la migración del proyecto.

GUT 7.2.0.

Plugin de **Godot** para escribir y controlar las pruebas de los scripts. Instalable desde la propia interfaz de Godot (tutorial aquí). Más información en el apartado Desarrollo dirigido por tests (TDD).

1.4.2. Trello

Se utilizará la plataforma **Trello** para coordinar todo el desarrollo y el flujo de trabajo del equipo siguiendo la *Metodología Kanban*. Más información en el apartado El Tablero Kanban.

1.4.3. GIT y Github

El código estará alojado en un repositorio GIT en **Github** en la siguiente dirección: `https://github.com/polirritmico/Bakumapu`. Todos los participantes deben tener una cuenta con permisos.

El control de versiones se explicará en detalle en el apartado Repositorio.

1.4.4. Google Suite

Por conveniencia, tanto los archivos para la organización del proyecto así como los de diseño artístico del juego se realizarán principalmente en la suite de Google. Más información en el apartado Google Drive.

1.4.5. LaTeX

Para realizar el presente documento se ha utilizado LaTeX 2_ε (del paquete TeX Live 2021 y el paquete TeX Live Extra) en el editor **TeXstudio v4.1.2**. Como alternativa se encuentra la plataforma online Overleaf (gratuita con registro). En cualquier caso, un archivo TEX es simplemente un archivo de texto plano, por lo que para su edición cualquier editor de textos sencillo debería funcionar sin inconvenientes.

Para su conversión a HTML se utilizará un archivo de instrucciones **Makefile** (GNU Make) que utiliza **make4ht** en su versión 0.3g, **pdflatex**, **Tidy**, **perl** y comandos GNU estándar (como **sed**, **mkdir**, **cd**, etc.). Más información en el apartado El documento de Diseño técnico.

1.4.6. Aseprite

Se utilizará en su versión 1.2.25 para el trabajo en estilo pixel art de los sprites de tiles, props, animaciones, GUI y la gran mayoría de los elementos visuales del juego. Ejecutable compilado para Windows aquí.

1.4.7. Krita

En su rama 4.4 se utilizará principalmente como alternativa (no pixel art) para el diseño visual de los personajes, mapas, etc.

1.4.8. Inkscape

Se utilizará como herramienta de diseño vectorial en su versión 1.1.

1.4.9. Finale 2014

Se utilizará para toda la composición musical en formato MIDI y MUSX.

1.4.10. Ableton Live

Se utilizará la versión 10.1.4 para los MIX, masters y/o conversión de archivos MIDI a formato WAV estéreo y toda la producción de diseño sonoro del juego. Tomar nota en caso de utilizar VST específicos.

1.4.11. FFmpeg

El formato para los archivos de música, sonido ambiente y efectos será el OGG. Para su conversión se utilizará FFmpeg en su última versión estable (actualmente la 4.4) con el encoder `libvorvis`. Los OGG deberán ser de 192 kbps a 44 khz:

```
$ ffmpeg -i ARCHIVO.wav -acodec libvorbis -q:a 6 ARCHIVO.ogg
```

1.4.12. Twine

Se utilizará en su versión 2.3.16 para graficar y diseñar el árbol narrativo del gameplay. Se utilizará el formato SugarCube v.2.36.1.

2. Visión general del desarrollo y principios del diseño técnico

2.1. Principios de diseño

Se pretende generar un código sólido, flexible y eficiente, pero el énfasis en la primera etapa de desarrollo estará en la *flexibilidad*. En principio, la idea original de diseño es solo eso, una idea; a medida que se vaya obteniendo más información y *aprendamos más del proceso de desarrollo y los requerimientos del diseño* deberemos ir refactorizando el código[1] para obtener unidades más definidas, mejor alineadas a los requerimientos y sobre todo unidades fáciles de mantener.

Para ello se deberá implementar un sistema de pruebas por cada feature que se trabaje. Esto implica invertir el proceso de desarrollo escribiendo primero el test, dejar que falle y a continuación implementar la funcionalidad. El principio es concentrarse en las pruebas no en el debugging; aunque es ineludible, *no queremos estar arreglando bugs, sino evitarlos.*

Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento.

2.2. Desarrollo dirigido por tests (TDD)

El qué antes del cómo

Se comienza a trabajar por la fase roja escribiendo un test que supervise la implementación de la nueva funcionalidad. *La idea no es probar cómo funciona el código, sino verificar que se comporte de la forma que esperamos.* Dado que aún el código no ha sido implementado este test fallará. Es importante no saltarse este paso y dejar que el test falle (el test solo es útil en cuanto efectivamente pueda fallar).

A continuación en la fase verde, se trabaja en la implementación del código tratando de superar el test de la forma más directa y rápida posible;

[1] Reestructurar su estructura interna sin alterar su comportamiento externo.

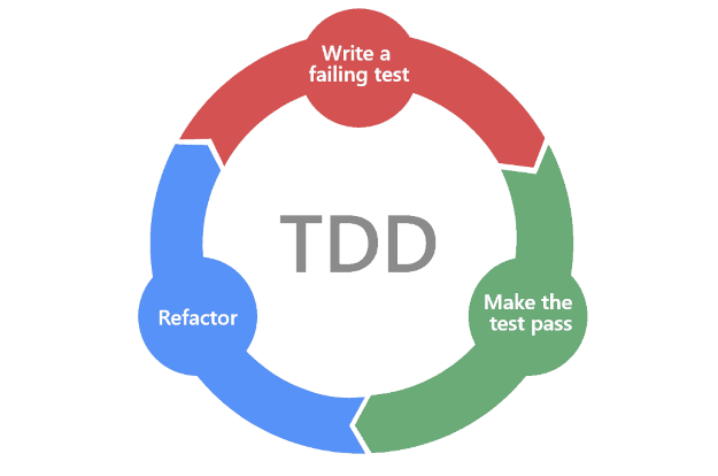


Figura 1: Flujo de desarrollo TDD.

no importa que el código sea sucio, débil o ineficiente.

Habiendo superado el test sabemos que contamos con un código funcional, pasamos entonces a la fase azul donde refactorizamos este código ahora en búsqueda de eficiencia y limpieza. Ya que contamos con nuestros test unitarios, podremos saber en todo momento cuando nuestra refactorización es correcta o genera problemas.

Repetimos el ciclo hasta alcanzar nuestras metas de implementación.

Siguiendo esta metodología, se otorgará mayor flexibilidad para el diseño creativo del juego. Al mismo tiempo, dada la magnitud y complejidad de todo el sistema, ya que el testeo del código es parte del proceso de desarrollo, se pretende minimizar considerablemente la cantidad de bugs que llegue a la rama **main** (ver el apartado Repositorio).

2.3. Refactorización: Rendimiento y limpieza

Aunque es frecuente limpiar el código a medida que se escribe, al hacer eso no estamos refactorizando; el factor clave de la refactorización es hacerlo intencionalmente pero de forma separada a la adición de nueva funcionalidad. *Al ya tener definidas las pruebas, la reescritura del código se puede hacer con seguridad y confianza.*

El objetivo en este proceso es doble, por un lado se busca un rendimiento óptimo en los apartados que lo requieran (combate, IA, físicas y animación por ejemplo) y por otro la revisión del código además de mejorar la legibilidad del mismo (mejores nombres de funciones, variables y comentarios), nos ayuda al entendimiento del sistema y puede señalar oportunidades de encapsulación y mejoras al modelamiento.

2.4. El documento de diseño técnico: una herramienta de diseño

El presente documento ha sido concebido como una potente herramienta de diseño, pues ofrece la oportunidad de reunir los elementos relevantes para la construcción de este software en un solo lugar ordenando y sistematizando el proceso de producción de forma coherente al proyecto.

Dado que Bakumapu se encuentra en estado de planificación o preproducción, el contar con un instrumento que reúna todas las dimensiones relevantes al desarrollo en un solo lugar, hace de su redacción y lectura un ejercicio de diseño. La continua revisión del texto servirá como una oportunidad para mejorar y optimizar el software y la administración de los flujos de trabajo.

Los detalles con respecto al uso del documento se verán dentro del apartado El documento de Diseño técnico.

Aplicando simplicidad junto al desarrollo colectivo de la documentación y el software, nos aseguramos que a medida que el código crezca, el equipo conocerá de manera más profunda y con mayor amplitud el sistema completo y la naturaleza de sus interacciones.

3. Flujo de trabajo y control de versiones

3.1. Metodología Kanban

3.1.1. El Tablero Kanban

Trello será la herramienta principal para mantener la coordinación del flujo de trabajo a través de un **Tablero Kanban**. Para el desarrollo del proyecto dividiremos el trabajo en tareas y las iremos incorporando a este tablero dentro de columnas que vienen a representar nuestros procesos de producción.

La idea es anotar cada tarea del proyecto en una tarjeta y agregarla a la columna de la izquierda. Mientras se trabaja en la tarea, su tarjeta va avanzando de izquierda a derecha hasta llegar a la última columna.

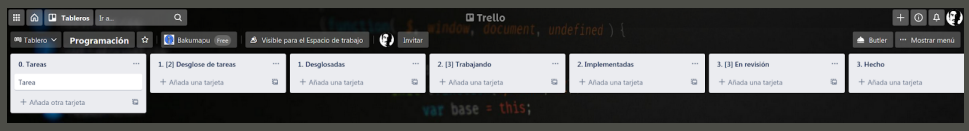


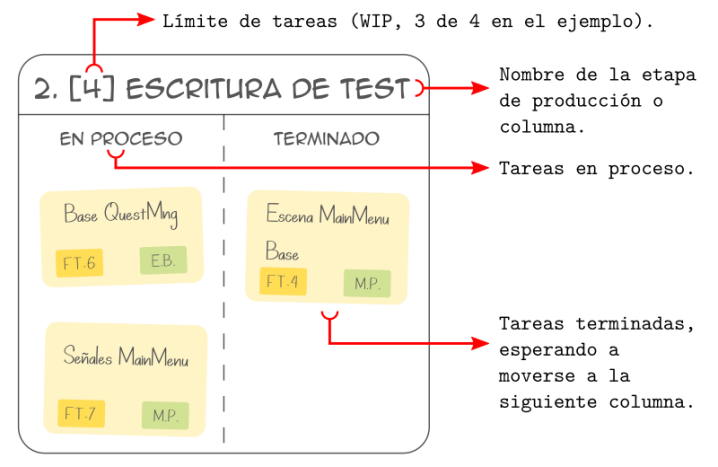
Figura 2: Tablero Kanban en Trello.

3.1.2. Límites de trabajo y columnas asociadas

Cada columna tiene asociado un límite máximo de tareas simultáneas (WIP), por lo que si el límite ha sido alcanzado no se podrán incorporar nuevas tareas a la columna hasta haber avanzado una de sus tarjetas a la columna siguiente. Este límite se puede ver en el número dentro de llaves [ ].

Al mismo tiempo cada columna tendrá una división vertical, dejando a la izquierda las tareas en proceso y a la derecha las tareas que ya han completado la etapa. El límite es compartido por ambas secciones, por lo que independiente de la sección en la que se encuentre la tarjeta no puede haber más tarjetas que el límite establecido en la columna. Ver Figura 3.

Figura 3: WIP y división de columnas.



Lamentablemente la plataforma Trello no permite dividir una misma columna en dos secciones, por lo que se utilizarán dos columnas distintas para un mismo proceso que llamaremos columnas asociadas. Para identificarlas se enumerarán todas las columnas y las asociadas mantendrán el mismo número. Por ejemplo en la Figura 4, la columna "[5] Trabajando" ha sido dividida en las columnas "2. [5] Trabajando" y "2. Terminadas". Ya que ambas comienzan con "2." sabemos que se refieren al mismo proceso y que sumando el total de tareas entre ambas no debemos superar las [5] (en este caso 4 en Trabajando + 1 en Terminadas; o sea ya hay 5).

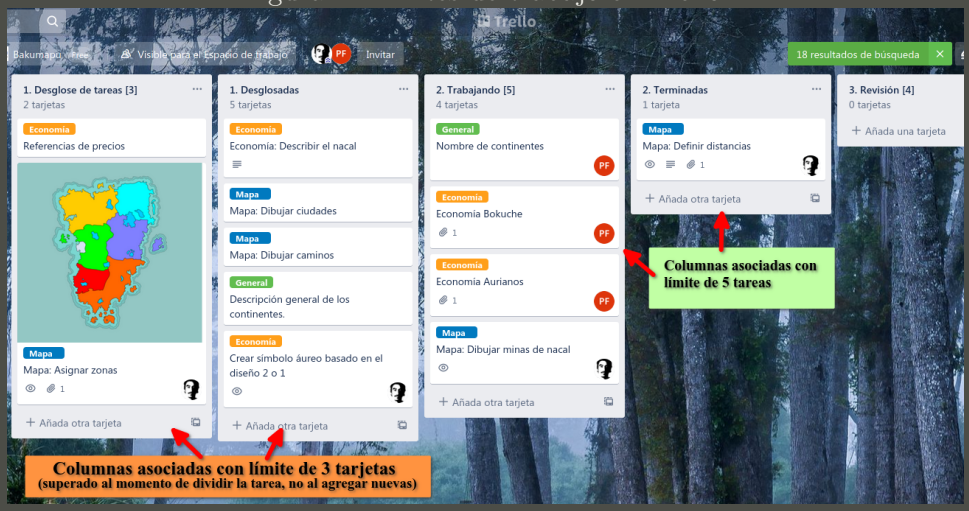
Respetar los límites de cada columna facilitará la distribución de trabajo y nos permitirá tomar decisiones más acertadas y realistas al planificar.

3.1.3. Descripción de las columnas del tablero

A continuación las columnas del tablero en orden de izquierda a derecha:

1. **0. Tareas:** Primera columna; acá dejamos las tareas en sus tarjetas y las ordenamos según prioridad (arriba las más urgentes). Solo se debería empezar a trabajar en la tarea superior. La prioridad de las tareas será determinada por el equipo.

Figura 4: Límites de trabajo en Trello.



2. **1. Desglose de tareas:** Las tareas que entran a esta columna se analizan y dividen en tareas acotadas. Es *muy importante* que las tareas sean descompuestas en tareas sencillas para que no tomen más de unos cuantos días en completarse.

3. **1. Ajuste de tarjeta:** A cada tarjeta en esta columna habrá que asignarle una ID (revisar Nombres de Tarjetas e ID), ajustar su nombre y su descripción de modo que quede perfectamente claro de qué se trata. Si es necesario, añadimos información relevante en la tarjeta.

4. **1. Asignación:** Aquí la tarea ya ajustada espera a que alguien la tome y se asigne a ella. Este encargado será responsable de completar la tarea hasta la columna **Implementadas**.

5. **2. Trabajando - Test:** En esta etapa escribimos el test y verificamos que falle (más información en el apartado Desarrollo dirigido por tests (TDD)).

6. **2. Trabajando - Código:** En esta etapa se implementa la funcionalidad hasta que se hayan pasado todos los test correspondientes y haya concluido el ciclo de refactorización con los test aprobados. Una vez

hecho el merge hacia develop en el repositorio central se pasa la tarjeta a la siguiente columna.

7. **2. Implementadas:** Acá quedan las tarjetas con tareas terminadas esperando (idealmente) la revisión de otro desarrollador, quien se deberá asignar a la tarjeta antes de moverla a la siguiente etapa.

8. **3. En revisión:** Acá van las tareas que se estén revisando. Si tienen problemas o bugs pequeños se resuelven aquí mismo, si tuviera bugs o un problema más grande podría volver a analizarse el desglose con alta prioridad (Columna 1. Desglose de tareas).

9. **3. Finalizadas:** Aquí van las tareas terminadas. Se archivan al hacer el merge de un conjunto de features completo a main (cuando sube la versión).

3.1.4. Flexibilidad del tablero

El tablero es una herramienta que permite visualizar las distintas áreas del proyecto, identificar cuellos de botella y contribuir a la gestión mediante límites de trabajo; no obstante, no se debe olvidar que *es solo una guía*. Si el flujo de trabajo cambia o el tablero entorpece algo, se debe ajustar rápidamente.

3.1.5. Nombres de Tarjetas e ID

Los nombres de las tareas se deben corresponder con la tarea o funcionalidad a implementar y a su rama asociada en GIT. Más detalles en el apartado Nombres de ramas y tarjetas.

3.2. Repositorio

El repositorio GIT de Bakumapu se encuentra alojado en la siguiente dirección: `https://github.com/polirritmico/Bakumapu`. Todos los participantes deberán contar con una cuenta con permisos de acceso y contribución.

3.2.1. Modelo de Integración continua

La contribución y el despliegue del código será organizada a través del modelo de Integración continua (Continuous Integration). Este modelo implica que *solo hay una versión del código que importa: la actual*. Para ello se trabajará principalmente en una misma rama que mantendrá una única versión del código de desarrollo. Esto implica que no podemos hacer commits solo cuando la feature esté completamente implementada, sino que debemos hacer commits continuos a medida que trabajamos en la implementación (idealmente varios commits en la misma jornada de trabajo). De ahí la importancia de tener un sistema robusto de test unitarios, de modo que cada commit haya superado todos los test locales.

Más información al respecto se puede encontrar en el apartado 4.

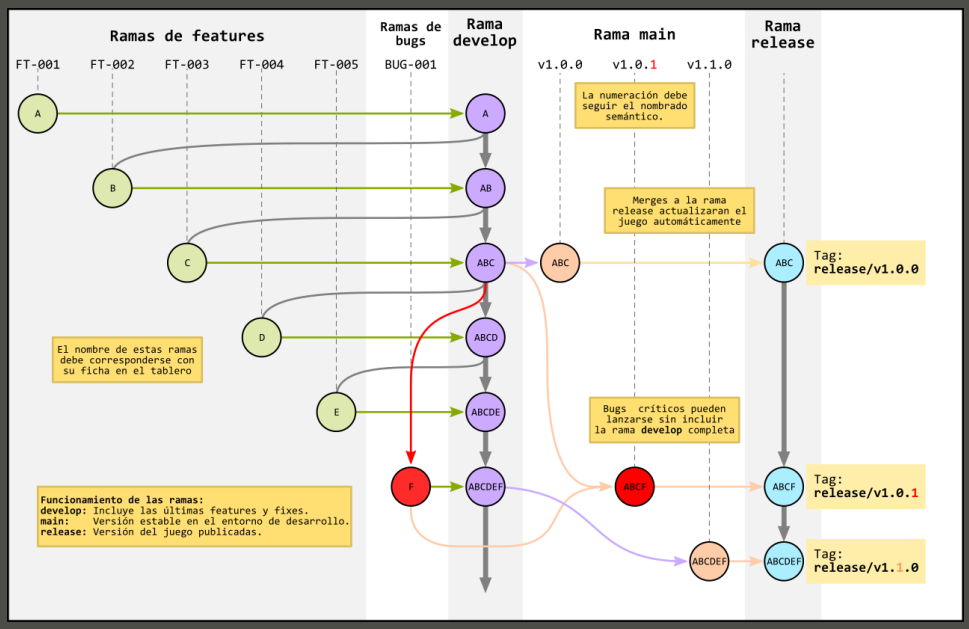


Figura 5: Ramas en GIT.

Ramas principales.
Bakumapu contará con una rama principal **develop** y 2 ramas de producción **main** y **release**:

a. **develop:** Es la rama principal del desarrollo ya que todo el código se escribirá permanentemente en ella. La idea es que cada desarrollador vaya haciendo continuos commits hacia **develop** de tal forma que los cambios en el código sean visibles para el resto del equipo y la implementación de una nueva funcionalidad pase el menor tiempo posible fuera de la rama. De este modo se facilitará la prevención temprana de bugs y problemas complejos de integración.

b. **main:** En esta rama se tratará de mantener el código lo más estable posible. Representa la última entrega del desarrollo y por lo tanto se creará a partir de **develop** cuando el código haya completado el conjunto de features de cada etapa de implementación del proyecto. Los testers trabajarán en base a esta rama.

c. **release:** Se creará a partir de **main** una vez que esta haya sido extensamente probada y depurada. Esta rama será la que pasará el código a producción por eso debe ser la más robusta (por ejemplo deploys automáticos a todas las copias instaladas y conectadas a internet).

Ramas de implementación Pese a que la mayoría de la implementación debería ser trabajada directamente en **develop**, para contadas excepciones se podría necesitar el uso de ramas separadas:

a. **Ramas de features:** Debe crearse una para cada funcionalidad que se quiera implementar a partir del estado actual de la rama **develop**. El nombre de la rama debe comenzar con "**ft-**" y corresponderse con el de su tarjeta asociada en el tablero (apartado Nombres de Tarjetas e ID). Se debe tratar de acortar lo más posible la existencia de esta rama de modo que si en determinado punto es posible hacer un merge hacia develop, se realice y se termine la implementación de vuelta en la rama principal.

b. **Ramas de bugs:** El nombre de la rama debe comenzar con "**bug-**" y corresponderse con el de su tarjeta asociada en el tablero (apartado Nombres de ramas y tarjetas). Al igual que en el caso de las ramas de features, la idea es trabajar el menor tiempo posible fuera de **develop**.

3.2.2. Commits

Todos los commits deberán tener al inicio del mensaje del commit la ID de la tarea correspondiente (ver el apartado Ramas de implementación y tarjetas).

3.2.3. Uso de GIT

Revisar el anexo Comandos de GIT.

3.3. Documentación

3.3.1. El documento de Diseño técnico

A nivel de desarrollo muchas veces terminamos dedicando más tiempo a estudiar el código y a entender su funcionamiento que a escribir nueva funcionalidad, por ello la documentación se vuelve tan relevante. Un código más fácil de entender ahorra tiempo, por eso ayuda a mejorar la estabilidad del software y en general todo el desarrollo se torna más productivo. La documentación técnica de Bakumapu estará dividida en este documento y en los archivos de código (apartado Formato y documentación dentro del código).

El presente texto, como ya se ha mencionado, tiene como objetivo desempeñar tres funciones principales:

1. Usarse como referencia ante dudas técnicas o de modelado.
2. Servir de instrumento de diseño.
3. Entregar toda la información relevante acerca del flujo de trabajo y del funcionamiento del software para integrar a nuevos miembros del equipo.

Para que estos objetivos se cumplan, el documento debe mantenerse actualizado a medida que se vaya escribiendo el código y tomando las distintas decisiones de diseño e implementación. Por lo mismo se ha generado un repositorio especial para ello (apartado Repositorio de documentación).

¿Qué se documenta aquí?

No se debe confundir la documentación de este texto con los comentarios o explicaciones dentro del código. El código debe estar debidamente documentado dentro de los archivos y líneas correspondientes (apartado Formato y documentación dentro del código). No obstante, cuando haya modificaciones importantes que involucren cambiar o definir la interacción entre clases o elementos de ámbitos más globales, se deberán anotar en este texto a modo de referencia técnica en el apartado Funcionamiento general.

Importante: Es muy relevante dejar en claro que el objetivo no es escribir una explicación *línea a línea* de cómo funciona el código, sino una *noción general* de ámbitos o lógicas más globales. Con señalar el sentido de estas entidades dentro del sistema y su interacción con el resto será suficiente.

3.3.2. Modificando el documento

Para modificar el documento, basta con utilizar un editor de textos sencillos y seguir la nomenclatura del sistema LaTeX. En el apartado LaTeX se proporcionará una breve guía con un listado de los comandos relevantes. En cualquier caso, considerando que el grueso del documento ya está definido, simplemente es cosa de usar los mismos apartados del documento como ejemplo.

Dado que en ciertos contextos la instalación de LaTeX puede ser bastante engorrosa (la instalación manual de muchísimos paquetes), no es necesario compilar una nueva versión con cada cambio sino simplemente mantener los archivos TEX y la versión dentro de **Makefile** actualizados.

3.3.3. Versionado del documento

Cada modificación a este documento deberá aumentar la numeración de la subversión en 1 (v0.0.**1** a v0.0.**2**). Los primeros 2 índices (v**0.0.**1) estarán en línea con la última rama **main** del repositorio. Cada vez que se suba una nueva versión de la rama, se deberá chequear que el documento contenga los cambios relevantes a esa versión e incorporarlos. Con cada cambio de versión menor, la subversión debe volver a cero (v0.**5.36** a v0.**6.0**). El versionado de las ramas se discutirá en el apartado Ramas principales.

Para ajustar la versión del documento solo hay que editar el archivo **Makefile**, que se encargará automáticamente de hacer todos los ajustes durante la recompilación:

```
Versión actual: v0.0.27.
```

```
Bakumapu-docs $ vim Makefile
```

```
SHELL = /bin/sh
# Actualizar con cada cambio
VERSION = 0.0.1
```

3.3.4. Exportar a HTML

Para exportar a HTML bastará con usar el paquete **make4ht** (utiliza pdflatex y htlatex). Las instrucciones de compilación están configuradas en el fichero Makefile, por lo que la conversión se automatiza con el comando:

```
Bakumapu-docs $ make html
```

Esto generará la página HTML con todos los archivos CSS, de fuentes y de imágenes relevantes en la carpeta /**docs**. Luego restaría simplemente actualizar el repositorio.

3.3.5. Repositorio de documentación

Un simple repositorio GIT en Gitbub, ubicado en: https://github.com/polirritmico/Bakumapu-docs. Desde aquí solo se puede revisar el código fuente del HTML, para acceder a una versión renderizada está la siguiente URL: https://polirritmico.github.io/Bakumapu-docs/.

Para sincronizar el servidor además de los comandos GIT habituales, el archivo **Makefile** tiene las instrucciones para automatizar el proceso:

```
Bakumapu-docs $ make sync
```

3.4. Formato y documentación dentro del código

La búsqueda de simplicidad en el diseño también aplica a la documentación del código. Idealmente éste debe estar "autodocumentado", es decir que los nombres de las variables, métodos y clases den cuenta de manera transparente e intuitiva su rol dentro de la lógica del algoritmo. En los casos más complejos, es de vital importancia añadir comentarios no solo para facilitar la comprensión de líneas más complejas, sino para ayudar al futuro proceso de refactorización y debbuging. *Los nombres largos no lastran la eficiencia del código.*

En cuanto al formato del código dado que GDScript está basado en Python, además de las propias sugerencias de Godot se recomienda seguir la guía de estilo PEP-8. En especial las siguientes indicaciones:

- Límite horizontal de 79 caracteres.
- Separación de 1 línea en blanco entre funciones y 2 entre clases.
- Indentación por 4 espacios.
- Operadores y variables separados por un espacio:

```
var ejemplo = Vector2(2, 5 + PI.get(2))
```

3.5. Google Drive

Se manejará la carpeta compartida Bakumapu, cuyo acceso será proporcionado a todos los miembros del desarrollo. En la raíz de esta carpeta se encuentran los documentos principales del diseño del juego, y las siguientes subcarpetas:

- **Diálogos:** Contendrá planillas con datos que serán importados a Godot programáticamente, es decir se deberá desarrollar un script o programa que transforme su contenido a XML, CSV o JSON y este sea manejable por Godot con *muy poca* o nada de intervención.
- **Gameplay:** Contiene el árbol de decisiones del gameplay y la información de las distintas quests.
- **Herramientas:** Contiene los instaladores, código fuente, o links de descarga del software mencionado en el apartado Listado de software y herramientas de producción además de los scripts de desarrollo. También contiene tutoriales para los colaboradores no técnicos del proyecto.
- **Locaciones:** Mapas y descripciones de los distintos lugares del Bakumapu.
- **Referencias:** Libros, imágenes, documentos, audios, videos y todo material referenciado para el desarrollo, inspiración, discusión o diseño del juego.

  Más información de estas herramientas en los apartados Cutscenes y diálogos y Quests. Además, estos archivos deberán seguir la convención de nombres detallada en el apartado Nombres de archivos.
- **Personajes:** Fichas de personajes.

3.6. LaTeX

Pese a lo que pueda parecer, el uso de LaTeX a nivel básico es bastante sencillo, simplemente es escribir el texto y utilizar comandos que definan el contenido para el compilador. En términos prácticos, básicamente es delimitar una sección, subsección o sub-subsección con el comando correspondiente y separar los párrafos con una línea en blanco entremedio.

En cualquier caso, los mismos archivos TEX del documento son un buen modelo de referencia. Si se quiere agregar información o apuntes para facilitar el uso de LaTeX se debería agregar en este apartado. En lo posible tratar de no agregar nuevos paquetes.

Para una lista de los comandos más comunes y una rápida explicación al respecto, revisar el anexo Comandos LaTeX.

4. Entrega continua

4.1. Flujo de entrega

En esta sección se abordará el sistema de deploy y el pipeline del proyecto.

5. Internacionalización

Primero revisaremos en términos muy generales la internacionalización y la localización.

5.1. Qué entendemos por localización (l10n)

Se entiende por l10n la adaptación del software con el objetivo de adecuarlo a las necesidades lingüísticas, culturales o incluso legales de un mercado, país o localidad en concreto.

Más allá de considerar la traducción de la documentación y la interfaz de usuario, la l10n es bastante más compleja pues además implica ajustar:

- Formatos numéricos, de fecha y de hora.
- Uso de símbolos de moneda.
- Uso del teclado.
- Algoritmos de comparación y ordenamiento.
- Símbolos, íconos y colores.
- Texto y gráficos que contengan referencias a objetos, acciones o ideas que en una cultura dada puedan ser objeto de mala interpretación u ofensas.
- Diferentes exigencias legales.

5.2. Qué entendemos por internacionalización (i18n)

El principal sentido de la i18n es diseñar el código de tal forma que el programa sea fácil de localizar y distribuir internacionalmente. Para ello es importante considerar los siguientes aspectos:

1. **Diseño y desarrollo que facilite la localización o la distribución internacional:** Por ejemplo, usar el sistema Unicode para la

codificación de caracteres, usar tipografías compatibles, controlar la concatenación de cadenas y evitar que el código dependa de strings pertenecientes a la UI.

2. **Crear métodos específicos para la localización:** Añadir etiquetas para habilitar texto bidireccional, la identificación de idiomas (ISO-639) o hacer la UI compatible con el texto vertical y otras tipografías ajenas al alfabeto latino.

3. **Preparar el código para que se ajuste a las preferencias locales, lingüísticas o culturales:** Esto supone incorporar características y datos de localización predefinidos a partir de bibliotecas existentes o de preferencias de usuario, formatos de fecha y hora, calendarios locales, formatos y sistemas de números, ordenamiento y presentación de listas, uso de nombres personales y formas de tratamiento, etc.

4. **Separar del código los elementos localizables:** De este modo podrán cargarse o seleccionarse alternativas localizadas según lo que determinen las preferencias internacionales del usuario.

5.3. Implementación

Tal como se ha mencionado en el apartado Visión general del desarrollo y principios del diseño técnico), muchos aspectos del diseño se irán definiendo en la medida que se obtenga más información de los desafíos y los requerimientos del software.

Dada la enorme cantidad de texto que se espera del proyecto, es de vital importancia considerar desde el principio los desafíos de la i18n y l10n. Al mismo tiempo, dado el desconocimiento de nuestra propia implementación, no podremos restringir *a priori* la elección de un mecanismo a otro. Por lo tanto el enfoque debe tener las siguientes consideraciones:

5.3.1. Exploración

Durante el desarrollo de las distintas funciones que requieran strings y assets con texto visibles por el jugador (letreros, videos, mapas), será importante considerar algún sistema para el manejo de los texto por cada idioma que soportará el juego. En este sentido, al mismo tiempo que los strings sean almacenados de forma eficiente para el uso en la ejecución del programa, deberán facilitar la modificación continua de las cadenas como parte integral del desarrollo.

5.3.2. Codificación

Para la codificación de caracteres se usará el sistema Unicode bajo el formato UTF-8 en *todos* los textos de contenido del juego.

5.3.3. Estructura de los directorios de l10n

En la carpeta raíz del proyecto se encontrará la carpeta locales. Dentro de ésta habrá una carpeta por cada idioma según su código ISO-639. Dentro, se crearán las mismas subcarpetas y archivos manteniendo la misma estructura de directorios. La cuenta, o cantidad de archivos y subdirectorios debe ser la misma para todos los idiomas. El idioma base para el contenido del juego será el español; no obstante los nombres de los archivos y las carpetas se deben escribir en inglés siguiendo la nomenclatura propuesta en el apartado Nombres de archivos.

5.3.4. Diagrama del árbol de directorios

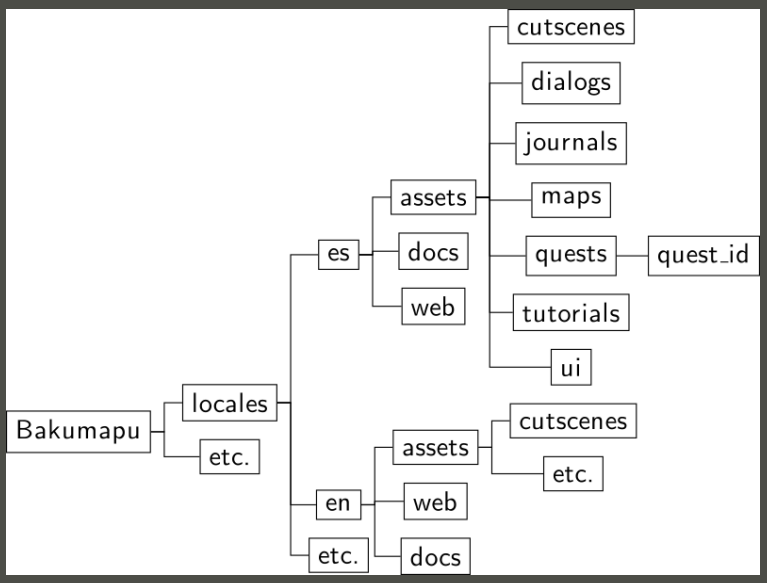


Figura 6: Organización de archivos de i18n.

5.3.5. Kit de desarrollo y API

Se estudiará la creación de un kit de herramientas para el desarrollo, específico de la l10n que interactúe con el programa y entre otras funciones permita:

- Generar documentos de referencia de personajes, locaciones, quests, etc.
- Generar reportes de actualización en el diseño o la l10n.
- Comprobar límites de strings, codificación y otras especificaciones.
- Comprobar nombres de archivos.
- Control de los elementos localizados pendientes.
- Categorización de cambios (relevantes, irrelevantes) y prioridades.

Referencias: El caso de BioWare.

5.3.6. Godot

Godot (versión 3.3) tiene implementadas dos formas de localización: a través de archivos CSV y mediante el uso de **gettext** (reducido).

Según este análisis previo a la implementación, el primer método es insuficiente y puede generar problemas al considerar la enorme cantidad de strings que debería contener el archivo CSV por cada idioma. Al ser un único archivo propone varias complicaciones al desarrollo tales como: el control de versiones, el trabajo en paralelo, algoritmos de parsing, corrupción y otros problemas.

Por otro lado, **gettext** parece ser una alternativa bastante acertada pues en principio permitiría cumplir todos los requisitos presentados en este apartado dependiendo la implementación del manejo de strings del código. Lo mejor de esta alternativa es que es uno de los estándares utilizados en los entornos de desarrollo de software, por lo que es una solución muy robusta y terminada. Además es importante destacar que cuenta con plataformas online tales como Transifex o Weblate y mucho soporte y documentación en general.

El mayor punto en contra de **gettext** es una curva de aprendizaje mucho más inclinada, pero por otro lado los desafíos de la i18n y la l10n son de por sí muy complejos. El utilizar herramientas creadas específicamente con estos fines puede ahorrar mucho tiempo en el largo plazo y al mismo tiempo mejorar la calidad del contenido.

5.3.7. Toma de decisiones

Dado que aún no hay código implementado la decisión de usar un sistema como **gettext** o implementar una API propia, se puede tomar más adelante. Lo importante es que en el diseño narrativo, se utilicen templates que permitan a futuro la utilización de scripts para acondicionar los formatos una vez se tenga decidido el sistema y su implementación.

5.3.8. Otras referencias

1. Video Game Internationalization: Best Practices for Localization Friendly Architecture
2. Godot: Localization using gettext
3. Gettext or Database translation?
4. The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets

6. Input/Output

El sistema interactuará con el usuario a través de inputs y outputs específicos, pero de momento no es necesario implementar tecnologías en concreto, sino mantener la flexibilidad del código y estudiar la propia implementación para decidir con mejores indicadores. Estas decisiones serán de responsabilidad del equipo de diseño en mutuo acuerdo con el equipo técnico.

A continuación se presenta un listado con elementos recurrentes, incluyendo algunos que desde ya se deberían considerar para que su implementación sea lo más fluida posible.

6.1. Inputs

Fundamentalmente son 3 elementos de entrada los que interactúan con el sistema a nivel de usuario: **Archivos** de carga, guardado y de configuración, **periféricos** como teclado, ratón, gamepad o touchscreen; e **internet** para actualizaciones y servicios en la nube como savegames y configuraciones.

A nivel de desarrollo se mantienen los mismos inputs, pero adicionalmente se incluirían archivos de stats, niveles, diálogos, sprites, audio, etc.

6.1.1. Archivos

Savegames: Archivos de carga a checkpoints u otro sistema de puntos de control dentro del gameplay. Se debería definir exactamente que se carga y que no. Por ejemplo, poder cargar el árbol de decisiones, stats, ítems, equipamiento, munición, etc., y no la posición de los NPC, ítems aleatorios, enemigos, entre otros.

Configuración: Permite cargar la configuración de usuario en distintas instalaciones o dispositivos. Este archivo debería ser un texto sencillo.

6.1.2. Controles

Teclado: Uno de los controles principales de juego. Lo importante es considerar la i18n para el mapeo y no utilizar teclas específicas de una layout en concreto; *no todos los teclados son* QWERTY.

Mouse: Requiere de un cursor que se adapte a los distintos contextos del juego y sprites especiales o efectos para resaltar el foco de los elementos visuales con los que interactúa el jugador. Por ejemplo: áreas de ataque, elementos del HUD, letreros, mapa, etc.

Si fuera necesario, se podrían agregar entradas en el menú de configuración para ajustar la aceleración, sensibilidad, velocidad del puntero, etc.

Gamepad: El mapeo del control deberá poder ajustarse en todo momento. Además, debería ofrecerse al usuario una forma sencilla de ajustarlo y el sistema debería reconocer automáticamente los modelos más comunes y ofrecer layouts predefinidas. El control debería ofrecer respuesta gradual para los sticks análogos.Por su parte Godot tiene una interfaz especial para los controles, por lo que para su implementación se podría comenzar por ahí.

Con respecto a las características adicionales de los controles más modernos tales como giroscopio, micrófono, panel tácticl, etc., se estudiará su implementación en el diseño del juego; pero evidentemente todo con muy baja prioridad en las primeras etapas.

Touchscreen: Se mostrará un gamepad virtual en la interfaz. Además se estudiará si mejora la jugabilidad interactuar directamente con elementos en el área de juego. Por ejemplo, para iniciar un diálogo con un NPC, se podría mover con el stick virtual y pulsar el botón de acción o simplemente presionar sobre el personaje y automáticamente el jugador camina hacia donde corresponde y abre el diálogo.

Touchpad: Similar al ratón. Investigar si es necesaria alguna consideración

especial.

6.1.3. Internet

Archivos de usuario: Lo principal a nivel de usuario es poder importar los ficheros de configuración y savegames desde algún sistema en la nube para facilitar la posibilidad de jugar una única campaña a través de diferentes dispositivos.

Actualización: Automáticamente el sistema debería poder comprobar la versión instalada del juego, informar la disponibilidad de una nueva versión, descargar los archivos necesarios e iniciar el proceso de actualización automáticamente. Además se deberán implementar opciones para desactivar tanto la descarga como la instalación automática de las actualizaciones.

Revisar la documentación de la API Steam Cloud.

6.2. Outputs

6.2.1. Controles

Vibración: El juego deberá soportar la vibración de controles y smartphones; idealmente con las distintas posibilidades que ofrece la tecnología como intensidad, duración, dirección, patrones, etc.

Touchscreen: En los smartphones la interfaz deberá mostrar los controles y alguna respuesta visual cuando se presione algún botón. Por lo mismo va a necesitar una interfaz especial para este tipo de dispositivos.

6.2.2. Video

El formato de salida de video será:

- Proporción: **16:9**.
- Resolución: **720p** y **1080p**.
- Framerate: **60 FPS**.
- Orientación: **landscape** (apaisada).

El uso de tecnologías como HDR o mayores framerates deberá estudiarse, pero probablemente no sean necesarias.

6.2.3. Audio

La codificación para los archivos de música, sonido ambiente y efectos será:

- Formato: **OGG** vorbis[2].
- Sample Rate: **44 khz**.
- Bitrate: **192 kbps**.

Los audios de sonido ambiente y música deberán ser estéreo, no obstante los efectos de sonido podrán ser monofónicos pues Godot ofrece un sistema de mezcla y posicionamiento interno que incluso permite mezclas en 5.1.

En cualquier caso, la implementación base será en estéreo y el soporte de mezclas de más canales deberá ser estudiado en base a un criterio técnico que considere la dificultad, el tiempo extra requerido, el impacto en el rendimiento del programa y retrocompatibilidad con equipos estéreo.

[2] En comparación al MP3, el OGG ocupa menos espacio y tiene mejor calidad a igual bitrate. En comparación el uso de CPU es marginalmente mayor, pero en una máquina moderna (más de 300 MHz) no debería suponer ningún problema.

6.2.4. Archivos

Partidas guardadas: El sistema debería poder generar rápidamente archivos de guardado que incluyan toda la información relevante para continuar la partida. Estos archivos deberán estar codificados para evitar trampas y se debe determinar qué tipo de información almacenan. Por ejemplo, se podría conservar solo la información con respecto al árbol de decisiones, stats, ítems, etc. y no a las posiciones de NPC, enemigos, ítems aleatorios, etc.

Configuración: Cualquier opción distinta a las por defecto deberían estar señaladas en un archivo de texto plano que contenga comandos y valores separados por el símbolo "=" de todas las opciones modificadas. Estos archivos serán útiles para guardar los ajustes del usuario entre distintos dispositivos y para realizar tests, ofrecer soporte y debugging.

Depurado: Quizás sea conveniente generar un mecanismo de generación de reportes con información relevante para facilitar el soporte a los usuarios, o facilitar la comprensión de los distintos fallos que puedan ocurrir sobretodo por consideraciones especiales en determinados sistemas.

Varios: Probablemente a medida que avance el desarrollo aparezcan nuevos archivos que sea conveniente generar. Por ejemplo: logs, informes de estadísticas del gameplay o algún tipo de telemetría[3]. A medida que estos archivos sean definidos deberían agregarse en este apartado.

6.2.5. Internet

Archivos de usuario: Lo principal a nivel de usuario es poder guardar los ficheros de configuración y savegames en algún sistema en la nube. Dependiendo del sistema utilizado se debería realizar una implementación acorde; por ejemplo, Steam tiene API específicas al respecto. También

[3]Importante: Cualquier tipo de data del usuario debería ser enviada bajo su conocimiento y consentimiento. Revisar implicaciones legales.

se podría facilitar la posibilidad de subir pantallazos a distintas redes sociales.

Soporte: El sistema debería ser capaz de generar y enviar reportes de error con información relevante del sistema y ejecución para debugging. Si es demasiado complejo que contengan el backtrace, al menos un informe que indique bajo que circunstancias se produjo el error o algo al respecto.

6.2.6. Otros outputs

Estos son outputs de tecnologías específicas que no deberían implementarse hasta las etapas finales del desarrollo (si es que se implementan).

Sonido desde el control: Independiente a la posibilidad de algunos controles más modernos de funcionar como una "segunda tarjeta de sonido" donde conectar audífonos, algunos mandos incluyen un parlante que ayuda a la inmersión en distintos contextos de juego. Se debe estudiar las posibilidades que ofrece Godot al respecto cuando se trabaje en los ports para las distintas consolas.

Leds del control: Probablemente se deba implementar una funcionalidad que ajuste el color de los leds del control en base a los puntos de vida actuales del jugador (verde a rojo), y algunos flashes de daño y ataque.

Pantallas del control: De momento no se considera implementar alguna funcionalidad con esta tecnología.

6.3. DRM

En la etapa final del desarrollo se deberá implementar un sistema *no invasivo* de control de copia del juego. Quizás asociar un archivo que contenga alguna clave de cifrado única con un usuario o descarga en concreto. Los sistemas de distribución como Steam suelen ofrecer soluciones al respecto.

Revisar la documentación de las librerías del framework de Steam: Steamworks Documentation - Steam DRM.

7. Modelado del software

7.1. Arquitectura

La principal decisión de diseño es construir una arquitectura que permita modificarse a lo largo del desarrollo, para ello se dividirá la funcionalidad del software en Unidades Manager. Estas unidades además de delimitar scopes, irán construyendo en conjunto un lenguaje común de funciones y variables, acorde a la lógica de funcionamiento del sistema. Es decir, el nombre de las funciones y variables de cada manager tendrán coherencia externa generando idealmente una "narración" del funcionamiento del código.

Por ejemplo, `tyanka.gd` podría reimplementar funciones de PlayerMng:

```
@@ src/characters/player/tyanka.gd
PlayerMng.storeItem(Tyanka.item_bag, GM.getRewardFromCurrentQuest(\
                    PlayerMng.currentQuest.getPath()))
```

```
@@ src/game/game_manager.gd
def getRewardFromCurrentQuest(path) -> item:
    quest = QuestMng.getQuestFromPath(path)
    return quest.getReward(path)
```

A nivel de arquitectura más global, tendremos la posibilidad de ajustar la interacción entre managers, aislando sus cambios internos de su exterior. Es decir, cada manager operará como una estación de una cadena productiva cuyo funcionamiento interno es ignorado por el sistema, siendo lo único relevante sus inputs y outputs esperados.

De esta forma por ejemplo, podremos fusionar o separar el manager de data entre uno que administre la memoria y uno que interactué con una base de datos, podríamos separar el manager de música con el de sonido sin cambiar el código de los managers específicos que lo utilizan concentrándonos exclusivamente en ajustar las llamadas en GM.

Por último e igual de relevante, esta encapsulación de funcionalidad va a servir para enfocar de mejor manera los distintos test unitarios que se generen

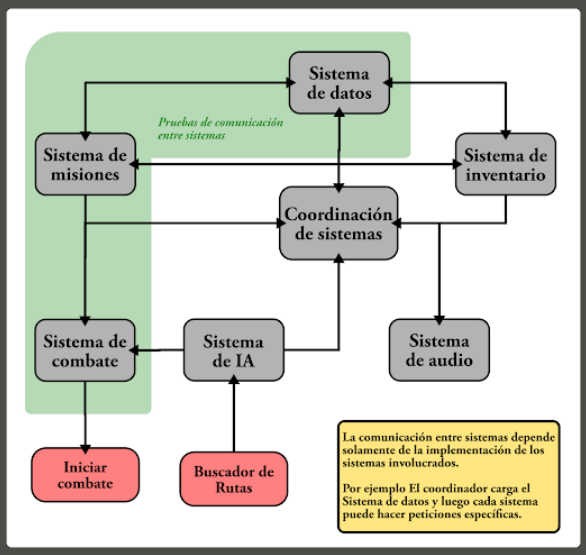


Figura 7: Ejemplo de sistema de managers ajustable.

en el desarrollo. Esto podría permitir agrupar una serie de comportamientos esperados, para comprobar relaciones entre los distintos managers (test de interacción) generando objetos y variables de casos o situaciones específicas.

En cuanto a su implementación, quizás el mayor desafío sea que la interacción con el modelo de Godot no lastre el rendimiento.

7.2. Funcionamiento general

Enfoque de diseño a priori

Como adelantamos, la idea principal es desacoplar los distintos niveles lógico-funcionales de diseño, de la implementación de los niveles y el resto del contenido.

Este sistema de managers encapsulará las distintas áreas del juego en una serie de submanagers y un manager principal que los coordine llamado GameManager (GM).

GameManager	
Manager principal a nivel de ejecución cuya función además de iniciar el juego será la de instanciar al resto de SubManagers. GM tendrá una clara tendencia a crecer en complejidad por lo que es muy importante que opere delegando funciones más que implementándolas. En esta misma línea si la interacción entre managers termina siendo demasiado compleja entonces será un síntoma de que es necesario un rediseño en cuanto a los in/outs de los submanagers implicados.	
SubManager	**Dimensión**
LevelManager	Todo lo relativo a los niveles. Por ejemplo, el conjunto de métodos que se encargan de toda la lógica relevante del nivel, una interfaz para que GM pueda conectar los distintos requerimientos del nivel, carga de locaciones, ubicación e inicio de subrutinas de Player y NPC, señales de áreas para CutsceneManager, UI específicas, etc. Al igual que GM posiblemente tenderá a complejizarse por lo que es importante estar pendiente de posibles desacoplamientos.
QuestManager	Todo lo relevante a las misiones: requerimientos, objetivos, experiencia, ítems, cutscenes, niveles, etc.
Cutscene Manager	Todo lo relativo a los momentos en el que el jugador no está interactuando con la UI ni tiene control del personaje, es decir animaciones dentro y fuera del gameplay. Incluye diálogos pero estudiar si es conveniente una separación.
PlayerManager	Todo lo relativo al jugador: estados, ítems, exp, estadísticas, estados de misiones, controles, path del árbol de decisiones, etc. Implementar una state machine.

IAManager	Todo lo referente a IA: Rutas de NPC, combate de NPC aliados y enemigos y si compete, la generación procedural de niveles.
UIManager	Todo lo relativo a la interfaz gráfica: Barras de vida, visualización del mapa, menús, inventario, árbol de habilidades e interfaz de diálogo.
AudioManager	Control de la música, transiciones, volumen, etc.; y los sonidos de los distintos objetos, su paneo, volumen, efectos, etc.
Debug Manager	Interfaz de debug que obtenga información relevante para el desarrollo sobre el funcionamiento de los distintos managers y objetos instanciados.

7.3. Diagrama del sistema

7.4. Funcionamiento del sistema de managers

Al partir, lo primero es iniciar GameManager que instancia los submanagers correspondientes para mostrar las escenas de presentación y luego iniciar el menú principal del juego. La escena del menú principal enviará señales a GM para que se inicie una nueva partida, se cargue o continúe partidas guardadas, se ajusten opciones, se inicien actualizaciones o se cierre el juego.

Independiente del sistema o mecanismo para iniciar o cargar una partida, será LevelManager el encargado de cargar los elementos del nivel como el mapa, los sprites, scripts de cutscenes, instanciar objetos, música, sonidos ambiente, quests, etc. Sin embargo será GM el encargado de conectar la lógica del nivel con los distintos managers correspondientes. Por ejemplo, LevelManager carga la música pero es GM quién conecta la señales de reproducción

con MusicManager. Es decir GM funciona como una interfaz a través de la cual LevelManager interactúa con el resto del sistema.

A continuación una idea general preliminar, o una guía para la implementación de los distintos managers:

7.4.1. GameManager

Descripción: Encargado de coordinar los submanagers e iniciar y controlar las distintas funciones de ejecución del programa (incluye la velocidad del juego).

Consideraciones: GM tendrá tendencia o a crecer o delegar en exceso, volviéndose inútil. La idea es que cuando un submanager necesite datos o funciones de otro submanager sea GM quien llame la función y no el propio submanager. De esta forma estaremos respetando la encapsulación del diseño y flexibilizando el código ya que en caso de desacoplar funciones de un submanager, no romperíamos la interacción entre sistemas y solo será necesario ajustar la llamada en GM.

Funcionamiento: Colección de métodos que accedan a data de los submanagers, cambien sus estados y/o informen de ellos. Controlar los estados del juego.

7.4.2. LevelManager

Descripción: Carga las escenas TSCN de los niveles y su lógica. Envía señales o ejecuta métodos para conectar las señales del nivel entre submanagers.

Consideraciones: También va a tender a crecer en complejidad, por lo mismo es importante delegar funciones específicas a los submanagers correspondientes y estar atentos a posibles rediseños en cuanto a su encapsulación.

Funcionamiento: Carga y almacena los niveles. Contiene funciones del tipo load_level(level). El objeto level podría contener el mapa, música,

áreas interactivas y objetos del nivel como props, NPC y player. La idea es que el sistema procese automáticamente todos los elementos relacionados al nivel.

7.4.3. QuestManager

Descripción: Controla el flujo de las quest en el gameplay, contiene los estados de las quests activas y actualiza el árbol de decisiones del jugador. Carga los distintos objetos o elementos de la quest en el o los niveles que corresponda. Dentro de estos elementos encontramos por ejemplo opciones de diálogo, ítems, áreas de interacción, NPC, etc.

Consideraciones: Cada quest es inicialmente un archivo de hoja de cálculo con todos los detalles de la quest que pasará por una herramienta del kit de desarrollo (apartado Kit de desarrollo y API) para importarse dentro de Godot como un nodo Quest o similar. La idea es que QuestManager sea capaz de interpretar estos nodos en tiempo de ejecución para simplificar los procesos de testeo y diseño. Es importante recordar que todos los métodos que trabajen con texto deben ajustarse a los métodos de l10n pertinentes (ver el apartado Implementación).

Funcionamiento: Extraer la data de los archivos quest referenciados por el nivel e incorporarlos al runtime del juego. Se debe mantener un control de las decisiones del jugador con respecto a las quests ya finalizadas y las que se encuentran en curso para modificar tanto quests, como diálogos disponibles según el árbol de decisiones u otras decisiones narrativas. El objeto Quest idealmente debería contener solo data y getter/setters. Sobre cómo se adjunta el objeto Quest a la colección de Quest activas, puede ser un comando específico en CutsceneManager o un objeto "interactible" que se adose a un área, un ítem, a una opción del diálogo, etc.

7.4.4. CutsceneManager

Descripción: Controla los momentos del juego en que el jugador no tiene el manejo directo de algún personaje o interfaz; se incluyen también los diálogos entre los NPC y/o el jugador. Cada cutscene corresponderá a un archivo de guión procesado por la herramienta del toolkit que la convertirá desde un formato de hoja de cálculo a un nodo dentro de Godot. Este nodo contendrá un listado de acciones secuenciales y la información requerida para su ejecución; será trabajo de CutsceneManager implementarlas y ejecutarlas. Las acciones van desde transiciones, fades a negro, diálogos, efectos atmosféricos, efectos de sonido, dar o quitar objetos; a mover NPC, agregar quests, cambios de estados de props, aparición de enemigos, etc.

Consideraciones: Lo principal es implementar un sistema de scripts que interprete las distintas instrucciones para los guiones adjuntos. La idea es que todas estas instrucciones se puedan ajustar directamente desde el archivo (o Godot). Puede ser útil añadir un sistema de control para que se ejecuten solo scripts que tengan satisfechas todas sus dependencias. Por ejemplo, indicar de antemano qué personajes u objetos deben estar implementados para que la cutscene funcione correctamente y arrojar un error en caso de que falte algo. Dado que trabajará con texto, no olvidar hacer los ajustes necesarios para la l10n del contenido (ver el apartado Implementación).

Funcionamiento: Cada nodo cutscene podría adjuntarse a un área de activación/interacción, a una entrada de diálogo, al tomar o activar un ítem, según condiciones de tiempo, quest completadas, etc. Al activarse la cutscene, CutsceneManager va interpretando y ejecutando las distintas líneas del guión según corresponda. Por ejemplo, para los diálogos cada línea podría tener una primera columna para identificar al personaje hablante, luego la siguiente que señale qué sprite usar (quizás asociado a alguna emoción del tipo enojo, alegría, herido, etc.).

Como se comentó en QuestManager, quizás sea útil un objeto "Interactible" que se pueda anexar a objetos relevantes como a un área de interacción, un timer, una transición, una opción de diálogo, etc.

7.4.5. PlayerManager

Descripción: Encargado de la conexión entre el objeto Player con el resto del sistema. Controla o informa los cambios en los estados de la Finite State Machine (FSM) del jugador, además de contener los distintos stats tales como experiencia, nivel, equipamiento, items de quest, munición, etc. Además, contiene el árbol de decisiones de quests y diálogos clave.

Consideraciones: Por el propio diseño que propone Godot, quizás mucha de estas funciones terminen siendo implementadas en la escena Player, sobre todo lo referente a la FSM. Para que el acoplamiento de ese código tenga un scope y relación coherente con los otros managers, es esencial estudiar la forma en que PlayerManager y la escena Player terminan interactuando entre sí y con el resto del sistema.

Funcionamiento: La escena Player podría tener nodos de sprite, FSM (con estados anexables), controles, animaciones, data y todo lo que interactúe directamente con los elementos visuales del juego. Por su parte PlayerManager debería controlar todo lo referente a la lógica del sistema de juego, vale decir los stats, modificadores de equipamiento al ataque, velocidad, árbol de decisiones, etc.

7.4.6. IAManager

Descripción: Encargado de conectar los métodos de IA con los distintos funcionamientos del juego, es decir los sistemas de inteligencia artificial de los enemigos en combate, el sistema de tácticas de aliados, pathfinding de NPC y si compete un generador de niveles procedural.

Consideraciones: Probablemente sea la dimensión más difícil de implementar, por lo mismo se propone no comenzar de inmediato con el

código sino investigar las distintas tecnologías disponibles y estudiar su implementación en Godot y dentro del sistema de managers del juego. Además debido a esta complejidad, lo más probable es que sea necesario utilizar un lenguaje de más bajo nivel como C++ para mantener una buena performance del sistema. Considerar la la división a submanagers más específicos.

Funcionamiento: Define en tiempo real las rutas de movimiento de los NPC, además de estrategias de ataque y posicionamiento tanto de enemigos como aliados en combate. Genera niveles en base a heurísticas de acoplamiento y optimización.

7.4.7. UIManager

Descripción: Controla toda la interfaz gráfica del juego y sus elementos como HUD, pantallas de inventario, equipamiento, menús del juego, escenas de diálogo, burbujas de texto, ayudas, letreros, mapas, información de debug, etc.

Consideraciones: Idealmente debe proveer una interfaz para que la información en la pantalla de los distintos managers se actualice cuando corresponda. Debe resolver un modo debug que muestre toda la información relevante al desarrollo. Si fuera necesario se podría delegar esta funcionalidad a un DebugManager.

Funcionamiento: Colección de escenas Godot que se muestran en función del estado del juego. Estas escenas envían señales a través de UIManager.

7.4.8. AudioManager

Descripción: Encargado de manejar todos los sonidos del juego incluyendo música y efectos. Debe controlar los efectos de sonidos (por ejemplo reverb en cuevas), paneos (por ejemplo en base a la posición en pantalla) y los fundidos entre canciones.

Consideraciones: Tratar que los distintos parámetros (duración de transiciones, ganancia, etc.) se puedan ajustar directamente desde la interfaz de Godot.

Funcionamiento: Quizás utilizar las animaciones de Godot para las transiciones. Los eventos deberían enviar archivos de audio y parámetros. Generar buses por escenario con efectos predefinidos (BUS de poco reverb, BUS de cuevas, BUS de escenas submarinas, etc.)

7.4.9. DebugManager

Descripción: Permite observar los diferentes estados de los distintos objetos de la ejecución, en especial aquellos relevantes para el control de flujo.

Consideraciones: Idealmente debe ser independiente al funcionamiento del juego y no debería intervenir directamente los distintos managers sino ocupar los IO disponibles de cada uno y no generar código específico dentro de los managers para debug.

Funcionamiento: Nodo adosable al objeto que queremos debuggear y que imprima en la consola de Godot o directamente en la pantalla la información relevante. Quizás algo parecido a la consola en algunos juegos tipo Fallout o Quake.

7.4.10. Otros managers

Como se ha comentado, quizás sea necesario agregar otros managers más específicos como por ejemplo: InventoryManager, ItemsManager, DialogueManager, SaveManager, etc. Lo principal es mantener la coherencia del sistema y mantener un diseño limpio.

7.5. Otras consideraciones

7.5.1. Plantillas

Algunos managers utilizarán objetos estandarizados; por ejemplo, QuestManager utilizará objetos Quest, CutscenesManager Cutscenes, AudioManager Music y Audio, LevelManager objetos Level, etc. La idea es generar plantillas de estos objetos para una expedita generación de contenido.

8. Organización del proyecto

Dado que el proyecto será manejado por personas en distintos sistemas operativos es vital mantener una coherencia en los nombres y la estructura de los archivos y directorios del proyecto. El presente apartado contiene una propuesta base que considera el funcionamiento general del desarrollo y la estructura de los archivos en el repositorio.

8.1. Organización de carpetas

- **Design:** Esta carpeta contiene todos los archivos del diseño de arte del juego (música, sonidos, sprites, tilesets, bocetos, dibujos, animación, etc.). Es de uso para artistas y desarrolladores y su organización se irá modificando en el tiempo. *Ningún archivo dentro de esta carpeta debe ser referenciado por el código*, por lo mismo permite mayor libertad para ajustar nombres y ubicaciones. Más información al respecto en el apartado Nombres de archivos.

- **addons:** Carpeta creada por GUT (tests) y otros plugins de Godot.

- **docs:** Contiene toda la documentación de librerías, licencias, API, SDK, manuales, ToDo, apuntes, notas de compilación y configuración relevante para el desarrollo del software.

- **export:** Contiene binarios para distintas plataformas del juego y librerías necesarias para su compilación (por ejemplo, librerías propietarias para generar los binarios de una consola).

- **locales:** Contiene todos los ficheros relativos a la i18n. Es la carpeta que utilizarán los traductores y desde donde el juego debe tomar todos los strings y assets correspondientes (assets con texto).

- **src:** La carpeta principal del desarrollo técnico. Además del código fuente del programa contendrá todo lo relativo a nodos, escenas, assets,

librerías. test y scripts. La estructura de esta carpeta será utilizada por el proyecto dentro de Godot.

Una consideración importante para mantener la organización de los archivos de una forma que sea coherente con la abstracción de *escenas y nodos*, es crear cada escena dentro de su propia carpeta. Aquí, además del archivo TSCN, deberían estar todos los assets que utiliza la escena para evitar cualquier problemas de dependencias. La idea es poder mover o copiar la carpeta de ubicación y que todo siga funcionando dentro de ella.

```
$ ls src/characters/player/tyanka/
animations/
    tyanka_walk_01.png
    tyanka_walk_02.png
    tyanka_jump_01.png
    etc.
portraits/
    tyanka_portrait_angry_01.png
    tyanka_portrait_angry_02.png
    tyanka_portrait_happy_01.png
    etc.
powers/
    animal_spirit.gd
    palin_shield.gd
    palin_whirl.gd
    etc.
tyanka.gd
tyanka.tsc
tyanka_stats.json
```

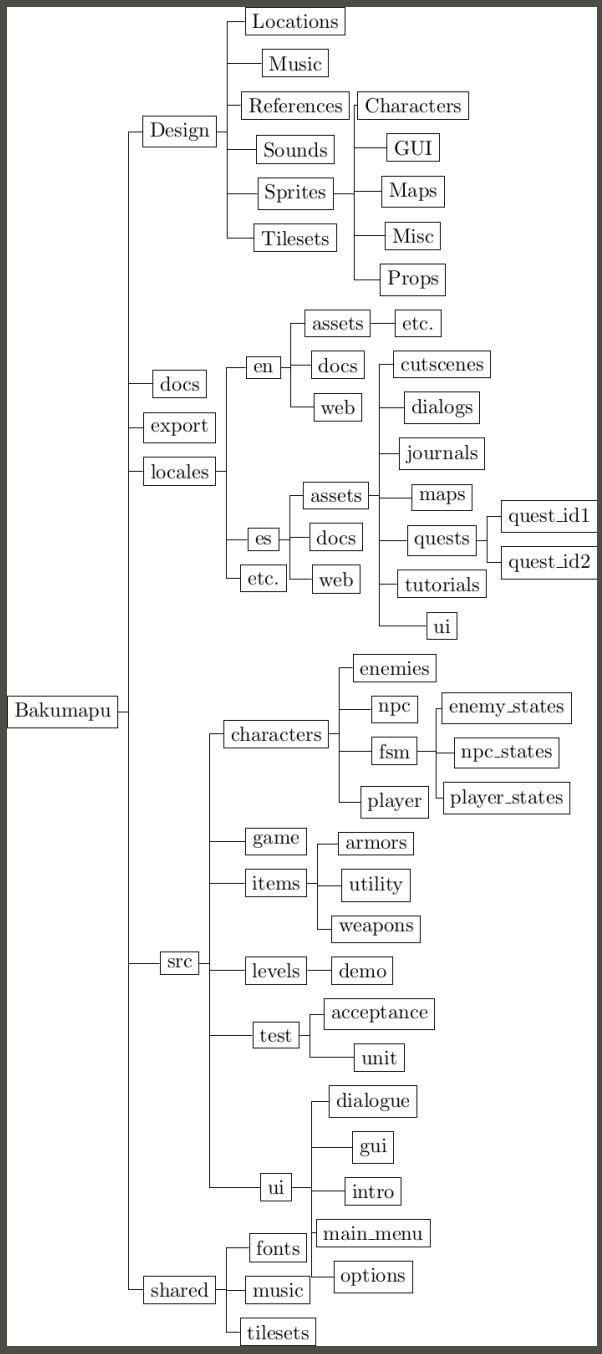
Si muchas escenas están utilizando los mismos sprites, probablemente se trate de archivos que deberían procesarse e ir dentro de la carpeta **shared** (detallada más adelante).

A continuación la organización preliminar de la carpeta **src**:

  - **characters:** Contiene las subcarpetas FSM, enemies, npc y player.
  - **game:** Todo lo relativo a los Managers, lógica y mecánicas del juego.
  - **items:** Todo lo relativo a ítems. A priori dividir en armors, utility y weapons.
  - **levels:** Escenas de niveles organizadas en distintas subcarpetas. Contemplar una ubicación para diversos templates.
  - **test:** Contiene carpetas con los distintos tipos de test: unit, acceptance, etc.
  - **ui:** Todo lo relativo a las escenas de la interfaz gráfica.

- **shared:** Colección de todos los assets y resources compartidos por más de una escena. Por ejemplo: fuentes tipográficas, música, tilesets, sprites, assets genéricos, etc.

Diagrama del árbol de directorios.



8.2. Nombres de archivos

En términos generales estas son las consideraciones importantes:

1. Los nombres solo podrán contener letras **a-z**, números **0-9** y guión medio "**-**" y bajo "**_**". No deberán contener acentos, ni eñe, ni espacios.
2. Todos los nombres de archivos deben estar en inglés.
3. Por compatibilidad, todos los nombres deben estar en minúscula y seguir el estilo snake_case (palabras en minúscula separadas por guiones bajos "_").

La carpeta Design tendrá mayor libertad en los nombres ya que los archivos no serán referenciados por el código; no obstante se sugiere seguir los mismos principios para evitar problemas de pérdida de datos.

8.3. Nombres de ramas y tarjetas

8.3.1. Ramas principales

La rama **develop** mantendrá siempre el mismo nombre. Por su parte las ramas **main** y **release** seguirán la nomenclatura del versionado semántico, en este caso **Bakumapu_vX.Y.Z** donde cada número representa:

- **X:** La versión principal. 0 alfa, 1 release. Si hay cambios no compatibles con la versión anterior, **X** se incrementa en 1.
- **Y:** La versión menor. Aumenta cuando se agreguen nuevas funcionalidades. Los archivos del usuario (savegames) deben ser siempre compatibles con la versión menor anterior.
- **Z:** La versión de patch. Siempre que se solucionen bugs se incrementa. Cuando avance la versión menor o principal vuelve a cero.

8.3.2. Ramas de implementación y tarjetas

Asignando una ID

Como ya se explicó en el apartado Flujo de trabajo y control de versiones, lo primero es anotar la tarea a trabajar en una tarjeta dentro del tablero Kanban. Cada tarea tendrá una ID específica que también se utilizará para nombrar la rama correspondiente en el repositorio. De esta forma será bastante sencillo hacer un seguimiento para depurar el código o cuando se evalúe el flujo de trabajo.

Esta ID tendrá 2 partes separadas por un guión "-". La primera, señalará el tipo de rama del repositorio asociada a la tarea:

- **Rama de features (ft):** Cuando se trate de implementar una nueva funcionalidad estamos hablando de una tarea que corresponde a las ramas de features y su nombre ID deberá comenzará con las siglas **ft-**.
- **Rama de bugs (bug):** Cuando se trate de la corrección de un BUG, las siglas serán **bug-**.
- **Refactorización (rfc):** Cuando la tarea se trate de refactorizar código ya implementado, las siglas a utilizar serán **rfc-**. La refactorización se debe trabajar directamente sobre develop y la ID debe usarse al momento de hacer el commit hacia develop.
- **Documentación (doc):** Los cambios en la documentación del repositorio del código usaran las siglas **doc-**. La nomenclatura no es necesaria para el repositorio de este documento.
- **Otros (misc):** Evidentemente el equipo puede evaluar en cualquier momento crear una nueva categoría sin inconvenientes. El único detalle es incluir esta categoría dentro de este mismo apartado. De momento para tareas puntuales se podría utilizar el identificador **misc-**.

La segunda parte de la ID será un contador n + 1 por cada sigla: ft-1, ft-2, bug-1, rfc-1, ft-42, etc.

* * *

Con estos sencillos elementos se pretende organizar el repositorio de tal forma que el flujo de trabajo vaya en línea con los criterios de gestión de una forma expedita y que permita a cualquier miembro del equipo hacer consultas directas al historial del repositorio para así facilitar el uso de herramientas de gestión y visualización de GIT.

8.4. Godot

- **Escenas:** Los nombres de las escenas dentro de Godot deberán seguir la notación UpperCamelCase.
- **Nodos:** Los nombres de los nodos parent seguiran la notación UpperCamelCase y para los nodos sin hijos la notación lowerCamelCase.

Ojo: Esto es para los nombres *dentro* de Godot *¡no para los archivos!*.

9. Kit de desarrollo de software

Tal como se ha descrito en el apartado Modelado del software, ya que el diseño del programa está orientado hacia la *flexibilidad en la creación y edición de los contenidos*, es necesario desacoplar lo más posible los elementos narrativos y de jugabilidad, de la programación.

Concretamente, se propone en primera instancia que todos estos elementos se trabajen desde hojas de cálculo o planillas en Google Sheets o similar, y desde allí se exporten a JSON, CSV o XML. Estos archivo serán convertidos a escenas (archivos TSCN) o scripts (archivos GDScript) e importados dentro de Godot.

Dado que la IDE de Godot permite exponer variables o miembros de una clase en el propio Editor de propiedades a través de la keyword **export**, estos elementos de diseño podrían ajustarse fácilmente tanto desde este panel como de su planilla. Esta posibilidad, muy útil para el ajuste rápido y fino de elementos de jugabilidad, puede al mismo tiempo ser una fuente de muchos errores.

Por ejemplo, si la velocidad de un enemigo se ajusta desde el panel dentro de Godot, estos valores no serán reflejados en la planilla correspondiente a no ser que se actualice manualmente la planilla. Si a esto agregamos la enorme cantidad de personajes, misiones, texto, etc. del contenido del juego, se hace evidente la necesidad de construir software específico tanto para automatizar la importación a Godot, como para mantener un control de estos elementos editados.

A continuación se presenta una propuesta de aplicaciones sencillas que faciliten todo este trabajo. Considérese que es una propuesta inicial y que se debe estudiar la externalización o no de determinados contenidos una vez se haya escrito y depurado la base del código. Así, estas herramientas ya estarán disponibles cuando comience la implementación de los contenidos, que a nivel de desarrollo se estima la de más trabajo.

9.1. Elementos narrativos

9.1.1. Textos varios

Se consideran textos como diarios, descripciones y pensamientos. Son elementos sencillos, básicamente strings. Como tal la transformación puede ser sencillamente a un nodo en Godot. Quizás el nombre del archivo podría indicar el tipo (entrada del diario, descripción, etc.).

9.1.2. Fichas de personaje

Además de servir para el proceso de traducción, es probable que ciertos strings como nombres, descripciones o incluso stats de altura y peso sean usados directamente por el software. Por lo mismo es importante tener un formato unificado al respecto. En este sentido quizás sea necesaria una herramienta específica que controle estos templates de personajes en base al funcionamiento de las escenas para poder importar sus strings (por ejemplo, el nombre o pseudónimo del personaje en distintos idiomas).

9.1.3. Cutscenes y diálogos

Básicamente es una hoja de cálculo. En la primera columna está el comando y en las siguientes los datos necesarios para que se ejecute. Probablemente deba transformarse a un archivo GDScript con una colección de strings.

Particularmente dentro de los comandos de la tabla de cutscenes, debiera existir la instrucción **dialog**. Como primer dato relevante para este comando, se debe señalar el personaje hablante, luego la "emoción" que indica qué sprite usar del personaje y finalmente la línea de texto en si. Revisar la información en el apartado CutsceneManager.

9.1.4. Quests

Esto va a depender de la implementación de QuestManager y Quest, pero a nivel de diseño de juego probablemente se trabajen en guiones o scripts similares a los de las cutscenes con comandos específicos en base a una planilla. Probablemente dentro de Godot debieran convertirse a nodos, archivos GDScript o escenas TSCN.

9.2. Elementos de mecánicas de jugabilidad

Todo lo relativo a las mecánicas o sistemas del juego, vale decir a qué velocidad se mueven los personajes, qué tipo de IA deben manifestar, cuanto daño produce la combinación de determinados ítems, el área de efecto, etc.

9.2.1. Estadísticas del jugador

Extraído de la planilla con los stats de personajes o de una hoja de cálculo general con todos los valores.

9.2.2. Armas, equipamiento e ítems

Muy probablemente tablas con stats. Un elemento por cada fila y un stat por cada columna. Estas planillas debieran poder exportarse y actualizarse de forma automática en Godot.

9.2.3. Personajes

Stats que se extraen de la ficha del personaje, o de una hoja de cálculo que contenga los stats de todos los personajes. Ver Fichas de personaje.

9.2.4. Locaciones o niveles

Quizás se pueda generar una escena base desde una planilla de texto para editar en Godot.

9.3. Traducción

Además de los elementos ya descritos, será muy útil disponer de software específico para la l10n del contenido del juego que permita o facilite llevar un control de la traducción y además generar documentación útil para los traductores.

9.3.1. Control de l10n

Ciertos elementos de la l10n van a necesitar un control de parámetros para evitar problemas del tipo cadenas de texto escapando fuera de la interfaz, diálogos demasiado rápidos, o los muchos errores con los textos y assets que pueden aparecer en el sistema. Para ello, esta herramienta podría mostrar un contador de caracteres que advierta la presencia de strings que superen el límite definido o realizar test de aceptación para todas las cadenas de diálogos, entradas de diario, quest, etc.

Un elemento muy importante a implementar es un mecanismo en la interfaz que informe la modificación de los strings base (del español) para poder ajustar la traducción. Se sugiere implementar check boxes que señalen strings ya traducidos y que se reseteen al haber cambios en el texto original.

También puede ser necesario un mecanismo para mostrar porcentajes de progreso o cuotas de avance en base a fechas de entrega, aunque quizás con el tablero sea suficiente.

9.3.2. Generación de informes o fichas

Es muy importante para una correcta i18n el contar con el material de escritura de los distintos elementos relacionados con la historia y la narrativa del juego (como fichas de personajes, descripciones de escenarios, mitología, trasfondos, etc.). Para ello, esta herramienta debiera poder mostrar y generar fichas con los elementos necesarios e informar al equipo de traducción cualquier cambio en los textos originales para que ajusten los elementos correspondientes.

9.3.3. Lista de requerimientos

Antes de comenzar a escribir el código de este programa, será necesario definir la lista de requerimientos en base a las necesidades del equipo de traducción y las especificaciones técnicas del formato de los documentos.

9.4. Lineamientos generales para el software del Kit

Para todas estas herramientas se propone utilizar Python, por su similitud con GDScript y por compatibilidad multiplataforma. Para la mayoría de programas solo será necesario una batería de scripts; no obstante, para los que use directamente el equipo de traducción se propone el uso de una GUI en Qt5 (PyQt5) por su versatilidad, facilidad de uso y licenciamiento.

Es relevante recordar que todos los textos de contenidos deben estar codificados en UTF-8.[4]

[4] Al parecer la codificación de los archivos TSCN debe ser US-ASCII. Investigar y testear antes de implementar la exportación a estos archivos TSCN file format.

10. Optimización

La optimización se debe definir avanzado el proyecto cuando se agreguen los elementos más complejos tales como pathfinding, IA, partículas, etc. De momento, salvo rendimientos desastrosos, la atención debería estar enfocada en escribir código limpio, legible y conciso.

10.1. ¿Qué optimizar?

El principio básico es: *no optimizar nada que no necesite optimización.* Para ello será necesario revisar la información del Debugger panel, el monitor de recursos y el resto de opciones de Godot para encontrar los elementos que lastren el rendimiento. En base a esa información se deberán tomar las distintas decisiones.

10.2. Referencias

- Debugger panel.
- Inferred types en Godot.
- GDNative C++ example.

A. Comandos de GIT

Este anexo es una referencia rápida a los comandos y configuraciones más comunes en el trabajo con GIT. Cuando en un comando dice origin es lo mismo que escribir la URL del repo en GitHub.
Otra referencia rápida que puede ser de utilidad se puede encontrar aquí.
Para información sobre el flujo de trabajo del repositorio ver el apartado Repositorio.

A.1. Trabajo local

- Ver el estado de la rama actual:

```
$ git status
```

- Cambiar a una rama:

```
$ git checkout rama
```

- Crear una nueva rama en base a la rama actual y cambiar a ella:

```
$ git checkout -b nueva_rama
```

- Agregar un archivo nuevo, editado o borrado al área de pruebas:

```
$ git add archivo
```

- Agrega todos los cambios al área de pruebas:

```
$ git add -A
```

- Agrega comentarios al commit en el área de pruebas:

```
$ git commit -m "A description with my changes"
```

  Para comentarios largos (más de 50 caracteres) usar:

```
$ git commit
```

Esto abre VIM en *normal mode* para editar el comentario. Pulsamos la tecla **i** para entrar al *insert mode*, escribimos los cambios normalmente y cuando terminemos pulsamos la tecla **esc** para volver al *normal mode* y luego **ZZ** para guardar y salir. Por el funcionamiento de GIT, el mensaje deberá tener esta estructura:

  - En la primera línea un HEADER de máx. 50 caracteres.
  - Segunda línea en blanco.
  - Las líneas siguientes contienen el body del mensaje con toda la explicación, detalles, listas, etc.

```
$ git commit
<Dentro de VIM pulsamos i para ingresar al insert mode>
First line as a HEADER of 50 char or less

Some extended description of all the changes:
- change 1
- change 2
- etc.
<Al terminar el texto, pulsamos la tecla esc y luego ZZ>
```

- Si queremos editar el mensaje del commit previo:

```
$ git commit --amend
```

- Mostrar diff de archivos en el área de staging:

```
$ git diff --staged
```

- Deshacer todos los cambios en base al último commit:

```
$ git restore .
```

- Borrar ramas locales ya eliminadas en origin:

```
$ git fetch -p
$ git remote prune origin
```

- Merges de ramas:

```
# Cambiamos a la rama de destino
$ git checkout rama-destino
# Actualizamos la rama de destino
$ git fetch
$ git pull
# Con todo preparado hacemos el merge
$ git merge rama-con-los-cambios
```

A.2. Acciones con el repositorio remoto

- Actualizar el repositorio local desde origin:

```
$ git fetch
```

- Actualizar y hacer merge al repo local desde origin:

```
$ git pull
```

- Enviar la rama a origin:

```
$ git push -u origin rama
```

- Envía los cambios locales de la rama actual a Github:

```
$ git push
```

- Borrar ramas:

```
# Borrar la rama local
$ git branch -d rama
# Borrar la rama en el repo remoto
$ git push origin -d rama
```

A.3. Configuración

- Clonar el repositorio en el espacio de trabajo local:

```
$ git clone https://github.com/polirritmico/Bakumapu
```

- Archivo de configuración de GIT:

```
$ ./config/git/config
```

```
conf[user]
name = USUARIO
email = MAIL_REGISTRADO_EN_GITHUB
[credential]
helper = store
[core]
autocrlf = input
```

- Credentials en archivo ∼/.config/git/credentials (asociado al usuario y al repositorio)

- Archivos y directorios ignorados por GIT (en .gitignore dentro de Bakumapu):

```
# Godot-specific ignores
.import/
export/
export.cfg
export_presets.cfg

# Mono-specific ignores
.mono/
data_*/
```

- Evitar problemas de formato de archivo entre Windows (CRLF) y Linux (LF):

```
$ .gitattributes:
```

```
# Set the default behavior, in
# case people
# don't have core.autocrlf set

* text eol=lf

# Explicitly declare text
# files you want to always
# be normalized and converted
# to native line endings on
# checkout.

*.godot text
*.tscn text
*.gd text
*.tres text
*.import text
*.md text
*.txt text
*.json text
*.xml text
*.py text
*.c text
*.h text

# binary files that should not
# be modified

# fonts
*.ttf binary
*.otf binary

# images

*.png binary
*.jpg binary
*.jpeg binary
*.webp binary
*.aseprite binary
*.gif binary
*.xcf binary
*.svg binary
*.kra binary

# sound

*.wav binary
*.ogg binary
*.sf2 binary
*.midi binary
*.amr binary
*.musx binary
*.mp3 binary

# misc

*.zip binary
*.rar binary
*.tar.gz
```

B. Comandos LaTeX

Los comandos son palabras claves que el compilador interpreta para generar el documento. Siempre deben comenzar con el signo backslash (\).
Para información con respecto al flujo de trabajo del Documento técnico, revisar el apartado Documentación.
A continuación una lista con los comandos más fundamentales:

1. Comentarios:

```
Esto es un texto normal.
% Esto es un comentario. No se compila.
Texto % Comentario.
```

2. Escapar símbolos:

```
Necesito escribir una linea \_ y un porcentaje \%
El signo \ funciona con este comando: \textbackslash.
```

3. Secciones (en orden descendente):

```
\section{Titulo de seccion}
\subsection{Titulo de subseccion}
\subsection*{Titulo.} % Asterisco para no ennumerar.
\subsubsection{Titulo de subsubseccion}
\paragraph{Titulo de seccion parrafo}
```

4. Ajustes tipográficos:

```
Negrita: Texto a \textbf{negrita}.
Enfasis: Texto a \emph{enfasis}
Cursiva: Texto a \textit{cursiva}.
Versalita: Texto a \lsc{versalita}.
```

5. Ajustes de párrafo:

   a) Alineación:

```
\begin{centering} % alternativas: flushleft, flushright
Texto alineado en base al valor en begin.
\end{centering}
```

b) Saltos de línea:

```
Los saltos necesitan este comando\\
sino se encadenan en una misma linea.

Los parrafos se separan con una linea en blanco.
```

c) Quitar sangría:

```
\noindent Primera linea sin sangria.
```

6. Referencias:

```
% Lo que queremos referenciar:
\subsection{Titulo subseccion}
% Le agregamos label. Ojala usar {nombre_archivo:descripcion}
\subsection{Titulo subseccion}\label{modelado:titulo-sec}

% Para crear la referencia hacia ese titulo:
La siguiente referencia: \nameref{modelado:titulo-sec}

% Ojo: En label{} solo ASCII basico, no tildes ni enie.
```

7. Listas:

```
\begin{enumerate} % Para listas no numeradas usar itemize.
    \item Primer item numerado.
    \item Segundo item numerado.
\end{enumerate}
```

8. Imágenes:

```
\begin{figure}[h] % h de here, t top, b bottom o nada.
    \centering
    \includegraphics[]{ruta/al/archivo}
    \caption{Subtitulo.}
    \label{fig:imagen} % Si queremos label para referencia.
\end{figure}
```

9. Código:

```
\begin{lstlisting*} % sin el asterisco
% Dentro de lstlisting solo ASCII basico sin tildes ni enies
var test = Clase.llamado(ejemplo.datos, objeto.pos())
$ comando -opciones ruta_a_archivo.txt
\end{lstlisting*} % sin el asterisco
```

10. Otros:

    a) Mover a la siguiente página si no hay espacio disponible:

    ```
    % el 2\baselineskip es para obtener el alto de 2 lineas
    \Needspace{2\baselineskip}
    ```

    b) Salto de página:

    ```
    \pagebreak
    ```

    c) Espacios indivisible (non-breaking space):

    ```
    60~hz, 50~km, siglo~\lsc{XVI}.
    ```

C. Trabajando...

C.1. Level Manager

src/levels/template/template_lvl.tscn Archivo de nivel base usado para el test

src/test/unit/test_level_manager.gd Script con el test de carga

src/test/unit/level_manager/level.tscn Archivo de test creado desde el nivel base para el test unitario.

El test inicia LevelManager, usa el método loadLevel() y usa los getters de LvlMng para extraer la data. Luego compara los datos obtenidos con los esperados.

C.2. Documentación

Pasar feature branch a rama develop

Underscore _ a funciones privadas

C.3. Modelación UI

Dialogue_manager dentro de la escena de diálogos, DialogueUI, en CanvasLayer

Todo lo relacionado a UI es muy propenso a cambio

No referenciar directamente con $ sino usar un export(NotePath) var _nodo

¿Singleton GameEvents para desacoplar GM?

Las clases interesadas en las señales se connectan en el ready hacia GameEvents. Por ejemplo, GameEvents tiene la señal "pelota_chocando", y dentro del objeto arco en ready conectamos la señal al método pelota_en_arco.

C.4. Dialogue_manager

Dialogos en formato **.tres**