Bakumapu

Diseño técnico.

Estudio ⅋ - 2021.8.2_v0.0.1

Índice

ToDo

P.

1. Introducción.

El presente documento busca ser una herramienta de diseño, un mapa de ruta que encamine tanto el desarrollo técnico del software en general como la implementación del código en el motor de juegos **Godot**. Pretende servir de ayuda o consulta para detalles específicos y servir de referencia para otorgar una visión general del funcionamiento interno del juego. Para facilitar el acceso a la información relevante 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.

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

1.1. Cómo usar este documento.

Lo ideal es una primera lectura para hacerse de una noción general del funcionamiento del software o lo que se espera lograr en cuanto a su diseño. Posteriormente la idea es usarlo a modo de referencia o mapa de ruta antes y durante la implementación del código.

El texto estará disponible en formato TEX, PDF y HTML en el repositorio de documentación del proyecto: `https://github.com/polirritmico/Bakumapu-docs`. La versión HTML se encuentra disponible en la siguiente dirección: `https://polirritmico.github.io/Bakumapu-docs/`. Si fuera necesario se puede generar una copia en formato Markdown utilizando Pandoc o algo similar.

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

1.2. Lista de requerimientos.

A continuación se presenta la lista inicial de requerimientos y consideraciones del programa. Importante destacar que este listado probablemente cambie a lo largo del desarrollo, además que las distintas implementaciones quizás generen nuevos requisitos o hagan obsoletos otros.

1. Enfoque en la facilidad de agregar y editar contenido.
    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 (1080 y 720 en PC).
8. Ajuste dinámico de la velocidad del juego.
9. Ajuste de stats de ítems.
10. Mixes de audio (stereo):
    a) Master.
    b) Música.
    c) Ambiente.
    d) SFX.

   e) Ataques.
   f) Props.
   g) UI.

11. Seguimiento de las decisiones en las quests (quests path).

12. States de guardado solo en locaciones del mapa.

   a) Ítems.
   b) EXP.
   c) HP y mana.
   d) Ubicación.
   e) Active Quests.
   f) Quest path.

13. Tácticas y equipo de la party.

14. IA de tácticas de la party.

15. IA enemiga.

16. Pathfinding de NPC.

17. Sistema de magia y poderes.

18. Sistema de partículas para ataques y locaciones.

19. Sistema de iluminación.

20. Interacción con props.

21. Dificultad ajustable en menú.

1.3. Listado de software y herramientas de producción.

1.3.1. Godot v3.3.

Se utilizará como herramienta principal de desarrollo el motor de videojuegos multiplataforma open source **Godot** en su rama 3.3 estable. Ante el lanzamiento de la versión 4.0 se estudiará la migración del proyecto.

GUT 7.1.0.

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).

1.3.2. 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.

1.3.3. 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.

1.3.4. 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.

1.3.5. LaTeX.

Para realizar el presente documento se ha utilizado LaTeX 2_ε (del paquete TeX Live 2021) en el editor TeXstudio v3.1.2. 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 **make4ht** en su versión 0.3g. Si es necesario convertirlo a Markdown se recomienda tomar el mismo HTML y transformarlo con Pandoc. Más información en el apartado El documento de Diseño técnico.

1.3.6. 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.

1.3.7. Krita.

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

1.3.8. Inkscape.

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

1.3.9. Finale 2014.

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

1.3.10. 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.

1.3.11. LAME 64bits.

Conversor de WAV a MP3 en su versión 3.100. El formato para los archivos de música y sonido ambiente será el MP3 en estéreo, con un bitrate de 192 kbps a 44.1 khz. Muy importante: conservar los archivos WAV.

```
$ lame -b 192 -m s ARCHIVO.wav ARCHIVO.mp3
```

1.3.12. FFmpeg.

Se utilizará en su versión 4.4 para convertir archivos WAV a OGG.

```
$ ffmpeg -i ARCHIVO.wav -acodec libvorbis ARCHIVO.ogg
```

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 esta primera etapa de desarrollo será hacia 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 particularmente, que sean 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 que queremos evitarlos.*

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

2.2. Desarrollo dirigido por tests (TDD).

Se comienza a trabajar por la fase roja escribiendo un test que supervise la implementación de la nueva funcionalidad. *No queremos probar cómo funciona el código, sino probar 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.

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; *no importa que el código sea horrible.*

Habiendo superado el test sabemos que contamos con un código que

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

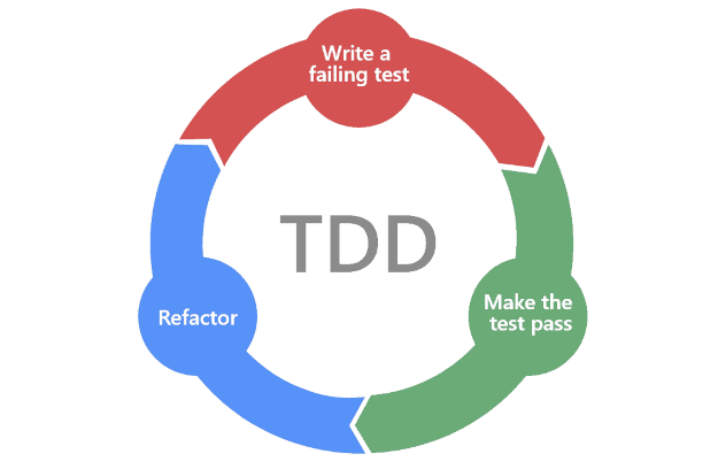


Figura 1: Flujo de desarrollo TDD.

funciona, 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 la gran cantidad de elementos relevantes para la construcción de este software en un solo lugar y por lo mismo, ordenar y sistematizar el proceso de producción en una manera coherente al proyecto y sus requerimientos.

Dado que Bakumapu se encuentra en estado de planificación o preproducción, el contar con un instrumento que reuna 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 optimización y mejora en el software y en la adminsitración de los flujos de trabajo.

Los detalles con respecto al uso del texto se verán en detalle 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.

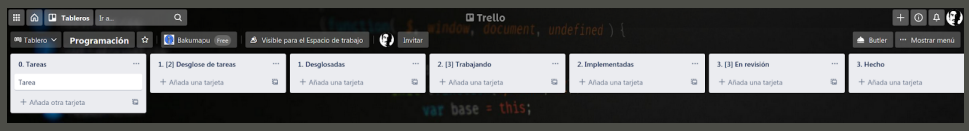


Figura 2: Tablero Kanban en Trello.

3.1.2. Límites de trabajo y columnas asociadas.

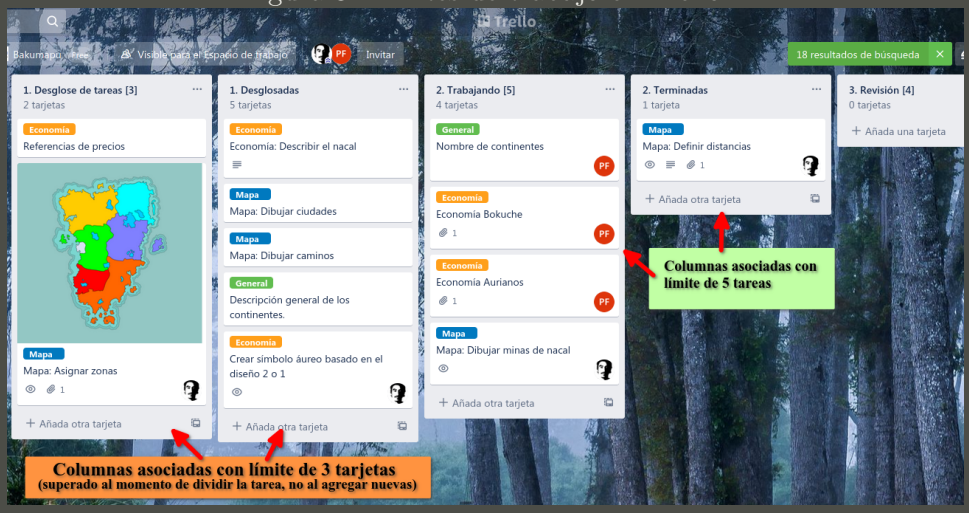
Cada columna tiene asociado un límite máximo de tareas simultáneas (WIP) por lo que si el límite ha sido alcanzado no se podrán incorporar nuevas tareas a la columna hasta haber avanzado una de sus tarjetas a la columna siguiente. Este límite se puede ver en el número dentro de llaves [ ].

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.

Lamentablemente la plataforma Trello no permite dividir una misma columna en dos secciones, por lo que se añadirá una enumeración al nombre

de cada columna que se repetirá en los casos de columnas asociadas. Por ejemplo, sumando las tareas de la columna "2. [5] Trabajando" y las de la columna "2. Terminadas", solo puede haber 5 tareas simultáneas por lo que no deberemos añadir una sexta.

Figura 3: Límites de trabajo en Trello.



Respetar los límites de cada columna, facilitará la distribución de trabajo en un modelo de entrega continua y nos permitirá tomar decisiones más acertadas y realistas al momento de la planificación.

3.1.3. Descripción de columnas.

El funcionamiento de cada columna es bastante sencillo:

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

2. **Columna Desglose de tareas:** Es *muy importante* que las tareas sean descompuestas en tareas sencillas para que no se demoren más

de unos días en completarse. Las tareas que entran a esta columna se analizan y dividen en tareas acotadas.

3. **Columna Desglosadas:** Aquí las tareas desglosadas esperan hasta que alguien se haga cargo de implementar el código y pasan a la siguiente columna.

4. **Columna Trabajando:** Aquí el encargado se anota en la tarjeta y trabaja hasta completarla, pasándola a Terminadas.

5. **Columna Implementadas:** Acá quedan las tarjetas con tareas terminadas esperando revisión.

6. **Columna 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.

7. **Columna Hecho:** Aquí van las tareas terminadas. Se eliminan al hacer el merge de un conjunto de features completo a develop (cuando sube la versión).

3.1.4. Flexibilidad.

Esta es una herramienta que permite tener visualizadas las distintas áreas del proyecto, identificar cuellos de botella y contribuir a la gestión mediante límites de trabajo; no obstante, no debe olvidarse que es solo una guía. Si algo del sistema necesita cambiarse porque va contra el flujo de trabajo, se debe ajustar rápidamente.

3.1.5. Nombres de Tarjetas.

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`.

La contribución de código será organizada a través del modelo de ramas de función (feature branching) adaptada a un sistema de TDD de entrega continua (más información en el apartado Principios de diseño).

3.2.1. Modelo de Ramas de función.

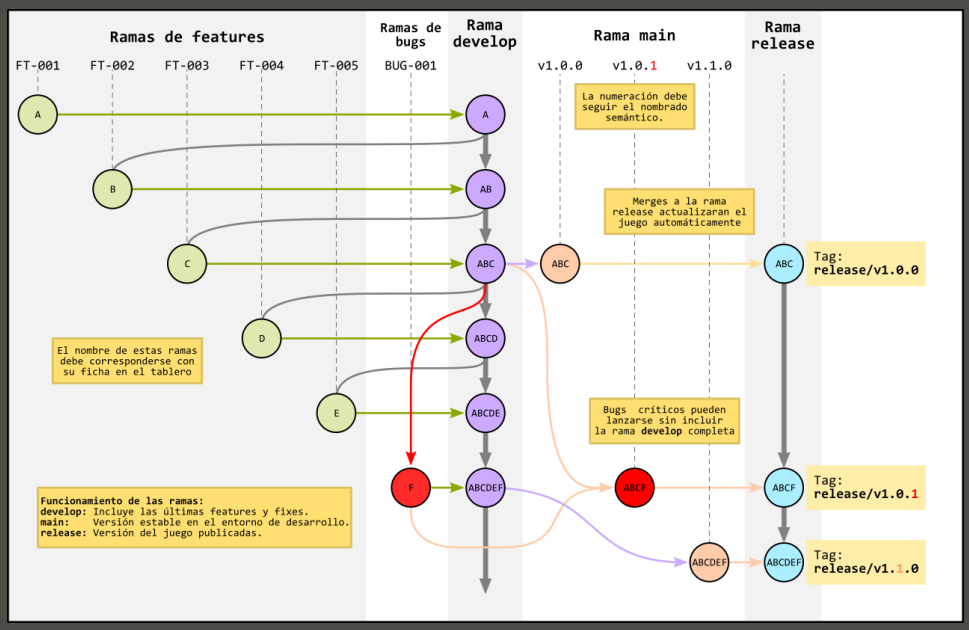


Figura 4: Ramas en GIT.

Ramas principales.
Bakumapu contará con 3 ramas principales, **develop**, **main** y **release**:

a. **develop:** Esta rama contará siempre con las últimas funciones implementadas. Es la rama principal del desarrollo ya que para trabajar cada feature, se creará una rama a partir de **develop**. A medida que se avance esa implementación continuamente se harán merges de vuelta a **develop** (al menos uno al día)[2]. De esta forma se pretende que esté siempre disponible el código completo para no perder la referencia global del proyecto.

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 **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.

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). La idea es desarrollar test que permitan verificar el funcionamiento de la feature y que deben ser superados antes de dar por concluido el trabajo en la rama. De esta forma se intentará hacer continuos merges hacia **develop** con la certeza de haber superado los test.

[2]Es muy importante este punto porque el tiempo que el código de la rama permanezca fuera de develop puede generar bugs desconocidos al estar aislado de los commit de otras ramas.

b. **Ramas de bugs:** Debe crearse una por cada bug encontrado. Dependiendo del bug la rama será creada a partir de **main**, **develop** o **release**. El merge del fix deberá ser hacia **develop** y si corresponde también a **release**. 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).

Para una información más detallada sobre el funcionamiento de GIT bajo el modelo de ramas de función ver el siguiente enlace (no olvidar los matices con respecto al trabajo bajo TDD en el apartado Visión general del desarrollo y principios del diseño técnico).

3.2.2. Versionado y nombres semánticos.

Los nombres de las versiones seguirán la nomenclatura de 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. No debería ser mayor a 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 solucione un bug se incrementa y cuando avance la versión menor o principal vuelve a cero.

Esta nomenclatura se debe aplicar en las ramas **main** y **release**.

3.2.3. Alternativas al modelo de ramas de función.

Si el flujo bajo el sistema de ramas de función resulta demasiado tedioso o la visualización general del código se pierde con mucha facilidad, ya sea por merges muy complejos o espaciados, se propone como alternativa la metodología de Entrega continua.

3.2.4. Comandos de GIT.

Este apartado es una referencia rápida a los comandos y configuraciones más comunes en el trabajo con GIT.

- Configuración:

```
$ ./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 y el resto de sistemas:

```
$ .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
```

- Clonar el repositorio en el espacio de trabajo local:

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

- - Ver el estado de la rama actual:

```
$ git status
```

- Cambiar a una rama (por ejemplo develop):

```
$ git checkout develop
```

- Crear una nueva rama en base a la rama actual (ejemplo “ft-01”:

```
$ git checkout -b ft-01
```

- Enviar la rama local a Github (origin es un alias de la ruta al repo en github):

```
$ git push -u origin ft-01
```

- Agrega todos los cambios al área de pruebas:

```
$ git add .
```

- Agrega comentarios al commit:

```
$ git commit -m descripcion
```

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

```
$ git push
```

- Actualizar el repositorio local desde Github:

```
$ git pull
```

Referencia rápida para comandos GIT aquí.

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; sus beneficios se pueden ver incluso a corto plazo. Un código más fácil de entender ahorra tiempo, por lo mismo 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 (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. Entregar toda la información relevante acerca del flujo de trabajo y del funcionamiento del software para integrar a nuevos miembros del equipo.
3. Servir de instrumento de diseño.

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 (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 (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, no es más que seguir simplemente el mismo modelo de 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. En este caso (cambio en la versión menor), la subversión debe volver a cero (v0.**5.36** a v0.**6.0**). El versionado de las ramas se discutirá más adelante en el apartado Versionado y nombres semánticos.

Al realizar estos cambios más grandes, lo importante es que después de editar el texto de la sección correspondiente, también se ajuste la versión dentro del archivo **Makefile**. Esto actualizará automáticamente la versión en todos los lugares correspondientes cuando se recompile:

```
Bakumapu-docs $ nano Makefile
```

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

Versión actual: v0.0.1.

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
```

3.3.6. 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))
```

3.4. 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:

- **Herramientas:** Contiene los instaladores, código fuente, o links de descarga del software discutido en este apartado además de los scripts de desarrollo. También contiene tutoriales para los colaboradores no técnicos del proyecto.

- **Referencias:** Libros, imágenes, documentos, audios, videos y todo material referenciado para el desarrollo, inspiración, discusión o diseño del

juego.

- **Historia:** Una especie de repositorio para la narrativa del juego. Contiene documentos sobre la historia, biografía de personajes, arcos narrativos, descripciones, locaciones, historia, trasfondos, etc.

  - **Quest y Diálogos.**

    Dentro de la carpeta Historia hay dos subdirectorios relevantes a nivel técnico: Quest y Diálogos. Cada uno 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* intervención.

    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.

3.5. 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 para marcar el estilo o contenido para el compilador. En términos prácticos, a nivel básico es escribir el título de una sección, subsección o incluso sub-subsección con el comando correspondiente y separar los párrafos con un línea en blanco entremedio.

En cualquier caso, teniendo los mismos archivos TEX del documento a modo de ejemplo y con la información discutida en este apartado, debiese ser suficiente para realizar todos los ajustes necesarios sin inconvenientes. Si hay algo que faltase y que pudiera aclarar el proceso, sería muy provechoso agregarlo al documento.

En lo posible tratar de no agregar nuevos paquetes.

3.5.1. Lista de comandos.

Los comandos son palabras claves que el compilador interpreta para generar el documento. Siempre deben comenzar con el signo backslash (\).
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:

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

4. Ajustes tipográficos:

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

5. Ajustes de párrafo:

   a) Alineación:

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

```
\subsection{Seccion a citar}\label{archivo:nombre-referencia}
Para facilitar el proceso en label ponemos el nombre del
archivo luego : y un nombre descriptivo. Soporta solo
caracteres en ingles, sin acentos ni letra enie.

Para crear la referencia hacia este titulo:
Mas info aqui: \nameref{archivo:nombre-referencia}
```

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
# Solo puede contener caracteres ascii basicos
# sin acentos ni enies.
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 \baselineskip es para obtener el alto de X lineas
    \Needspace{2\baselineskip}
    ```

    b) Salto de página:

    ```
    \pagebreak
    ```

4. Internacionalización.

4.1. 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 y/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 o contenido fuente de los elementos localizables:** De este modo podrán cargarse o seleccionarse alternativas localizadas según lo que determinen las preferencias internacionales del usuario.

4.2. 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 interfaz de usuario y de la documentación, la l10n es bastante más compleja pues además implica el ajuste de:

- 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.

4.3. 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:

4.3.1. 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.

4.3.2. Codificación:

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

4.3.3. 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.

4.3.4. Diagrama del árbol de directorios:

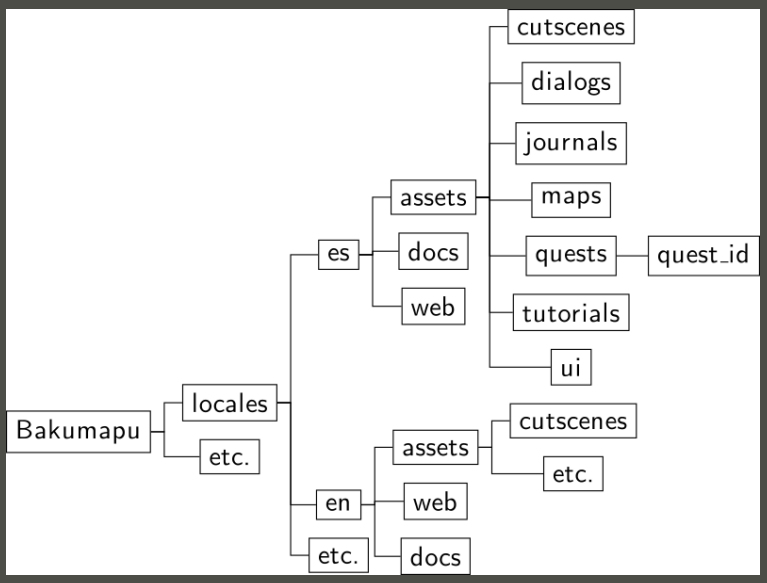


Figura 5: Organización de archivos de i18n.

4.3.5. 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.

4.3.6. 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.

4.3.7. 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.

4.3.8. Otras referencias.

1. Video Game Internationalization: Best Practices for Localization Friendly Architecture
2. Godot: Localization using gettext
3. Gettext or Database translation?

5. Modelado del software.

5.1. 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. De esta forma la flexibilidad de la arquitectura permitirá cambiar y generar lo que se espera de cada dominio del Manager.

Por ejemplo, tendremos la posibilidad de fusionar el manager de música con el de sonido, o separar el sistema de datos en un sistema que interactúe con una base de datos de idiomas y otro que parsee archivos de misiones, etc.

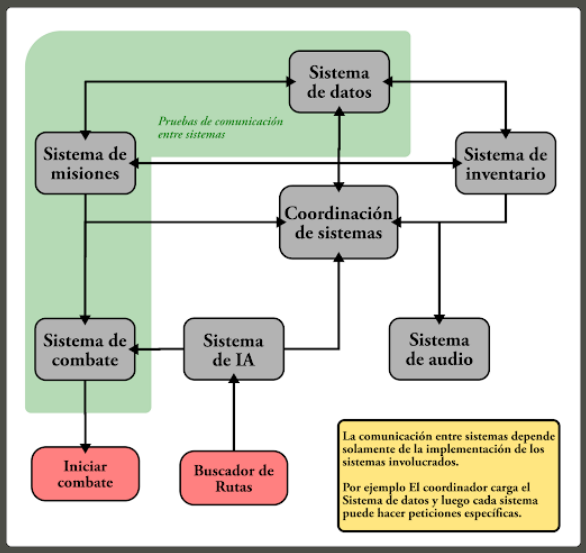


Figura 6: Ejemplo de arquitectura ajustable.

El enfoque agrupa los distintos comportamientos esperados a nivel de funciones al tiempo que favorece el surgimiento de una narrativa en cuanto a los nombres de las clases, funciones, variables y su sentido dentro del sistema global. Se trata de generar una especie de cadena de producción que opera en base a inputs y outputs esperados.

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 en el desarrollo. Esto incluso posibilita agrupar una serie de comportamientos esperados para generar objetos y variables de casos o situaciones específicas con el fin de comprobar interacciones entre los distintos Managers (test de interacción).

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

5.2. Sistema de Managers.

Enfoque de diseño a priori.

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

Este sistema de managers por tanto vendrá a funcionar como una especie de motor lógico dentro de la implementación en Godot. Para ello se encapsularán las distintas áreas del juego en una serie de submanagers (o cajas negras) 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. Los sprites de player implementarán 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.

5.3. Diagrama del sistema de managers.

5.4. Funcionamiento general.

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 cargar y conectar la lógica del nivel; esto es cargar el mapa, los sprites, scripts de cutscenes, instanciar objetos, cargar música y sonidos ambiente, asignar quests activas, etc. Es LevelManager quien lee la información o scripts de un nivel concreto e informa a GM para conectar estas señales con los submanagers correspondientes.

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

5.4.1. 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.

5.4.2. 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.

5.4.3. 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.

5.4.4. 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 planilla de datos 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). 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.

5.4.5. 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 FSM (Finite State Machine) 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 desde PlayerManager a GM y al resto del sistema.

Quizás mucha de la funcionalidad descrita termine siendo implementada en la escena Player en Godot sobre todo lo referente a la FSM. En ese sentido para que ese acoplamiento de código tenga un scope e interacción coherentes con el sistema, es esencial estudiar la forma en que PlayerManager y la escena Player terminan interactuando entre sí y con el resto de managers.

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.

5.4.6. 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.

5.4.7. 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 compete, se puede 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.

5.4.8. 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.

5.4.9. 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.

5.4.10. 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.

5.5. Otras consideraciones.

5.5.1. 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.

6. Input/Output.

1.ToDocio

Desde el punto de vista del usuario son básicamente 3 elementos: Archivos (como archivos de carga/guardado y de configuración), periféricos (como teclado y ratón, gamepad, o touchscreen); e internet para actualizaciones y servicios de cloud para savegames y configuraciones.

A futuro dentro del desarrollo será preciso implementar algunas tecnologías en concreto. De momento solo es necesario mantener la flexibilidad del código y estudiar la propia implementación para decidir con mejores indicadores.

7. Organización del proyecto.

Dado que el proyecto será manejado por distintas personas en distintos sistemas operativos, es vital mantener una coherencia en la estructura y el orden de los archivos y directorios del proyecto.

A continuación se presenta una propuesta base que considera tanto los ficheros del código como los del diseño artístico:

7.1. Árbol de directorios.

- **Design:** Esta carpeta contiene todos los archivos relacionados al desarrollo artístico de los distintos componentes audiovisuales del juego tales como música, sonidos, sprites, tilesets, bocetos, dibujos, animación, etc. Es de uso interno para artistas y desarrolladores, y debe mantenerse organizada bajo reestructuración regular en el tiempo. *Ningún archivo dentro de esta carpeta debe ser referenciado por el código*, por lo mismo permite mayor libertad en los nombres de los archivos. Más información al respecto en el apartado Nombres de archivos.

- **docs:** Contiene toda la documentación de librerías, licencias, API, SDK, 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 binarios de una consola).

- **locales:** Todos los ficheros relativos a la i18n, es decir la carpeta que utilizarán los traductores y desde donde el juego debe tomar los strings y assets correspondientes.

- **src:** Probablemente la carpeta principal para el desarrollo técnico. Además del código fuente del programa, contendrá todo lo relativo a nodos, escenas y scripts de Godot.

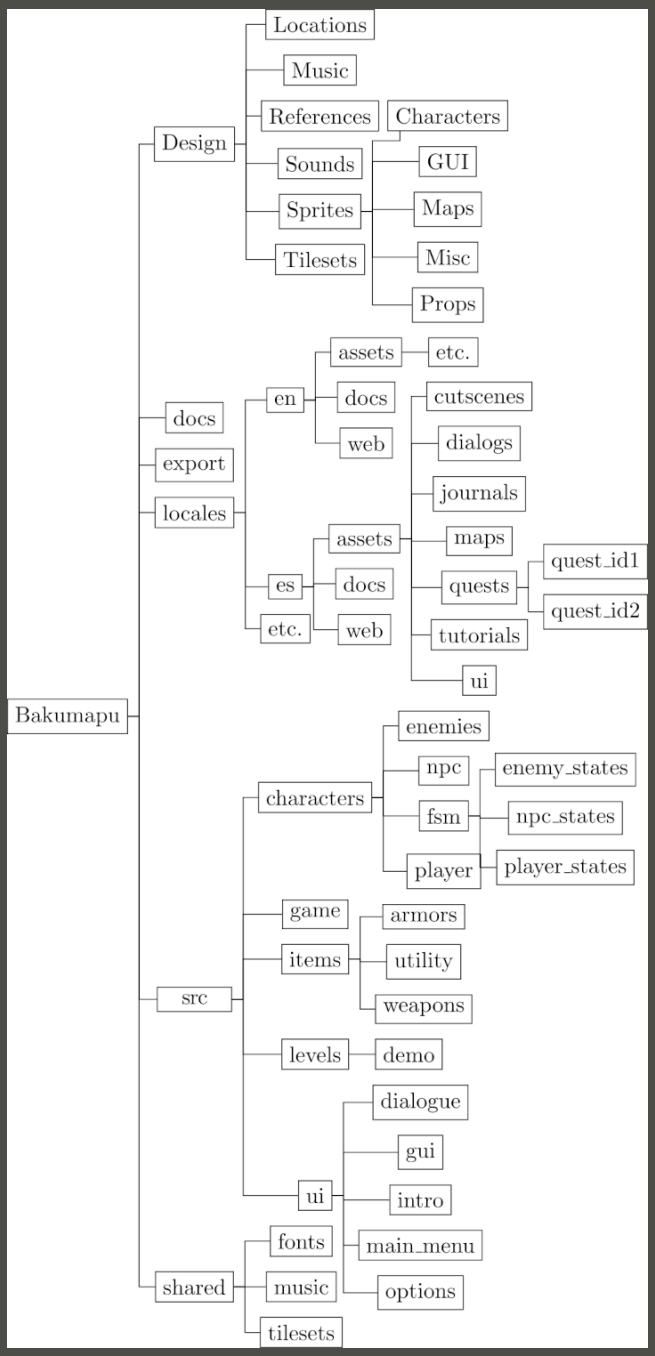
Debido a que ésta será la misma estructura que se usará dentro de Godot, es preferible que la carpeta de la escena, además de contener el archivo TSCN, contenga todo lo necesario dentro de sí misma para evitar problemas de dependencia.

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

  - **characters:** Contiene las subcarpetas FSM, enemies, npc y player.
  - **game:** Todo lo relativo a los Managers y el funcionamiento 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.
  - **ui:** Todo lo relativo a la escenas de 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.



7.2. Nombres de archivos.

Los nombres de archivos deben seguir ciertas especificaciones.

En la carpeta Design los nombres de los archivos pueden contener signos como eñes, acentos y mayúsculas.

7.3. Nombres de ramas y tarjetas.

Ya que las ramas del repositorio GIT y las tarjetas del tablero Kanban tienen su origen en la misma tarea designada por el equipo de desarrollo, para mantener un seguimiento más riguroso y tener mejor información compartirán un mismo nombre. Este nombre será designado al momento de establecer la tarea y deberá cumplir las especificaciones detalladas a continuación.

Los siguientes nombres corresponden a cada subrama:

1. Rama de features: FT-
2. Rama de bugs: BUG-
3. Rama de patches: PATCH- Los patches son cambios al código motivados por features nuevas más que por problemas.

Así, tareas relacionadas con algún elemento de la interfaz gráfica del juego tendrán el nombre de "ui-"

Las ramas con nombres con los que se va a interactuar son las de features y bugs. Las ramas de nombre fijo Hay tres tipos de subramas, ramas de features, ramas de bugs y ramas de patch

8. Kit de desarrollo de software.

Tal como se ha descrito en la sección Modelado del software del presente documento, ya que el objetivo está encaminado 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. Esto abre posibilidades de error pues 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. Si a esto agregamos la enorme cantidad 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 se debe estudiar la externalización o no de determinados contenidos una vez se haya escrito y depurado la base del código. Así, cuando comience la etapa de la implementación de contenidos —que a nivel de desarrollo se estima la que implica más trabajo—, estas herramientas ya van a estar disponibles.

Recordar también en cuanto a la implementación que los textos de contenidos deben corresponder a la codificación UTF-8.[3]

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

8.1. Elementos narrativos.

8.1.1. 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.).

8.1.2. 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).

8.1.3. 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.

Dentro de los comandos de la tabla de cutscenes, existe el de **dialog**. Junto a este comando en la siguiente columna está el personaje hablante, luego la "emoción" que indica qué sprite usar del personaje y finalmente la línea de texto en si.

8.1.4. 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 transformarse a nodos, archivos GDScript o escenas TSCN.

8.2. 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 determinada armadura y arma, el área de efecto, etc.

8.2.1. Estadísticas del jugador.

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

8.2.2. 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.

8.2.3. 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.

8.2.4. Locaciones o niveles.

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

8.3. Traducción.

Además de los elementos de jugabilidad y narrativos ya descritos, quizás sea muy útil disponer de software específico para la l10n del contenido del juego que permita por un lado, un seguimiento o control de strings y assets ya traducidos y los por traducir, en base a criterios de relevancia y límites de strings, dimensión o similares; y por otro lado la generación de informes con todo el material y documentación del transfondo de todo el mundo de Bakumapu. Así se conseguirá mejorar la calidad de la traducción y dinamizar el proceso con él o los equipos encargados de manera considerable.

Descripción detallada: como va a funcionar esto qué deberia entrar y que salir. Y una idea de como preparar el material de input pera que este programa o script sepa qué interpretar y generar estos documentos de traducción.

8.4. Lineamientos del software del Kit.

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

9. Optimización.

inferred types en Godot