MASTER

Bienvenido

Bienvenido a la documentación oficial de **MISI** (Motor de Inteligencia de Simulación Interactiva), el proyecto de simulación felina en Godot Engine. Este es un motor de juegos 2D y 3D gratuito y de código abierto impulsado por la comunidad.

> Nota: Esta documentación está en desarrollo activo y puede contener información que cambie en futuras versiones del proyecto.

La simulación MISI permite crear comportamientos realistas de gatos en entornos urbanos procedurales inspirados en Cusco, con IA comportamental avanzada y física realista.

MISI es una simulación que combina:

- **Comportamientos felinos dinámicos** que se adaptan al entorno sin animaciones pregrabadas
- **Generación procedural de ciudades** tipo arquitectura andina usando algoritmos geométricos
- **Sistema de física realista** implementado con IK (Inverse Kinematics) para movimientos naturales
- **IA comportamental** basada en máquinas de estado que responden al contexto del entorno

> Warning: Este proyecto requiere Godot 4.x o superior para funcionar correctamente.

ARQUITECTURA

Descripción de la arquitectura general del sistema MISI y cómo sus componentes trabajan en conjunto para crear una simulación coherente.

Estructura del Proyecto

El proyecto se organiza en **módulos especializados** que trabajan de manera coordinada, cada uno con responsabilidades específicas pero interdependientes:

```
cat_proj_test3/
├── scripts/
│   ├── Main/           → Sistema principal del gato
│   ├── CityHandler/    → Generación procedural de ciudad
│   └── HouseGenerator/ → Construcción de edificios
├── assets/             → Modelos 3D y recursos
├── materials/          → Materiales PBR
├── textures/           → Texturas y mapas
└── tests/              → Escenas de prueba
```

La carpeta `scripts/Main/` contiene todos los sistemas relacionados con el **comportamiento del gato, incluyendo su locomoción, IA, sistema de cámara y controles. **La carpeta `CityHandler/` maneja toda la **generación procedural de la ciudad** usando algoritmos como diagramas de Voronoi y gestión de chunks

espaciales. **La carpeta** `HouseGenerator/` se encarga específicamente de la **construcción de edificios individuales**, incluyendo techos, ventanas y detalles.

Flujo de Datos

Pipeline de Generación

El sistema genera la ciudad siguiendo un **flujo jerárquico descendente**:

- **SpatialChunkManager** gestiona chunks espaciales en tiempo real basado en la posición del jugador.
- **VoronoiGenerator** crea diagramas de Voronoi para dividir el espacio en distritos orgánicos.
- **ManzanaManager** administra bloques urbanos (manzanas) asegurando estabilidad geométrica.
- **HouseBuilder** construye edificios individuales con detalles procedurales como techos, ventanas y texturas.

Sistema de Comportamiento

El comportamiento del gato opera mediante un **sistema de estados coordinados**:

- **BehaviorHandler** gestiona los diferentes estados del gato (caminar, correr, saltar) usando máquinas de estado.
- **main.gd** coordina la locomoción física y la toma de decisiones del gato.
- **targets_cat.gd** maneja el rigging IK para movimientos naturales de la columna y extremidades.
- **Camera.gd** proporciona seguimiento dinámico con efectos visuales y controles intuitivos.

Tecnologías Utilizadas

El proyecto utiliza **tecnologías modernas** para lograr comportamientos realistas: **Godot Engine 4.x** como motor base con soporte completo para 3D y física. **GDScript** como lenguaje principal por su integración nativa y facilidad de desarrollo. **Algoritmos especializados** incluyendo diagramas de Voronoi para división espacial, IK para animaciones procedurales, y subdivisión de polígonos para lotes urbanos. **Sistema de chunks** para optimización con LOD dinámico basado en distancia. **CharacterBody3D** con detección avanzada de terreno para física realista del gato.

SISTEMA FELINO

Documentación del sistema principal de comportamiento del gato, el núcleo de la simulación que maneja toda la lógica de movimiento, estados emocionales y respuestas al entorno.

Controlador Principal

main.gd - Núcleo del Sistema

Este archivo contiene la **lógica principal del gato**, implementando un sistema de **locomoción procedural** que simula movimientos felinos realistas. El sistema **no utiliza animaciones pregrabadas**, sino que genera

movimientos dinámicamente basándose en el terreno y las decisiones de la IA. La locomoción se basa en un **sistema de estados finitos** que coordina el movimiento de las cuatro patas independientemente, permitiendo adaptación a terrenos irregulares.

El sistema implementa **detección de terreno en tiempo real** usando raycasting para que cada pata encuentre automáticamente el punto de apoyo más adecuado. Además, incluye un **sistema de cola dinámico** que responde al estado emocional del gato, agregando realismo comportamental. La **toma de decisiones** se basa en distancias, obstáculos y objetivos, permitiendo transiciones fluidas entre caminar, correr y saltar.

Estados de Locomoción

```
enum Estado {
    PASO_1, PASO_2, PASO_3, PASO_4, PASO_5,
    SALTO_PREP, SALTO, ATERRIZAJE,
    CORRER_PREP, CORRER_SALTO, CORRER_ATERRIZAJE
}
```

Los **estados de paso** (PASO_1 a PASO_5) implementan el **patrón de marcha felina** donde las patas se mueven en secuencia específica para mantener estabilidad. Los **estados de salto** manejan la preparación, ejecución y aterrizaje cuando el gato detecta obstáculos o desniveles. Los **estados de carrera** se activan cuando el objetivo está lejos, aumentando velocidad y longitud de zancada.

Parámetros Configurables

Movimiento Base:

- `VELOCIDAD_BASE`: Controla la velocidad general del gato, afectando tanto movimiento como rotación
- `VELOCIDAD_MOVIMIENTO`: Factor multiplicador para la velocidad de movimiento de patas específicamente
- `VELOCIDAD_ROTACION`: Velocidad angular para girar hacia objetivos
- `LONGITUD_PASO`: Distancia física que recorre cada paso, calibrada para proporciones felinas

Sistema de Salto:

- `MIN_HEIGHT_JUMP`: Altura mínima de obstáculo que requiere salto (vs. simple paso)
- `MAX_HEIGHT_JUMP`: Altura máxima que el gato puede saltar físicamente
- `UMBRAL_DETECCION_SALTO`: Distancia hacia adelante donde el gato detecta obstáculos

Parámetros de Correr:

- `DISTANCIA_PARA_CORRER`: Distancia al objetivo que activa modo carrera automáticamente
- `VELOCIDAD_CORRER`: Multiplicador de velocidad aplicado durante carrera

Sistema de Cola Felina

El **sistema de cola** es una implementación única que simula **movimientos felinos auténticos** basándose en el estado emocional y la actividad del gato. La cola **no es decorativa** sino que responde dinámicamente al comportamiento, agregando una capa adicional de realismo. El sistema utiliza **física procedural** con patrones

específicos para cada estado emocional, desde movimientos suaves y ondulantes hasta movimientos rígidos y alertas.

El sistema implementa **personalidad individual** donde cada gato desarrolla patrones únicos de movimiento de cola, incluyendo preferencias direccionales y intensidades diferentes. Los **estados emocionales** se determinan automáticamente basándose en la actividad actual (explorar, concentrarse, correr), y cada estado tiene su propio **patrón de movimiento matemático** que genera trayectorias realistas.

Estados Emocionales

```
enum EstadoCola { RELAJADA, CURIOSA, ALERTA, JUGUETONA, CONCENTRADA }
```

RELAJADA genera movimientos suaves en forma de "S" cuando el gato está tranquilo. **CURIOSA** produce ondulaciones más animadas con variaciones caprichosas cuando explora. **ALERTA** crea movimientos más rígidos y erguidos durante situaciones de atención. **JUGUETONA** genera patrones erráticos y divertidos (activable programáticamente). **CONCENTRADA** produce movimientos controlados con pequeños temblores durante preparación de saltos.

> Nota: El estado "JUGUETONA" es activado por el programador. No está activado en el producto final.

Funciones Principales

- `actualizar_cola(delta)` **Núcleo del sistema de cola** que actualiza posición y estado cada frame. Calcula la posición base usando el centro de las patas traseras, determina el estado emocional basándose en la actividad actual, aplica el patrón de movimiento correspondiente, y agrega la personalidad individual del gato. También maneja el cambio ocasional de "preferencias" para simular comportamiento felino natural.
- `patron_cola_relajada()` → Vector3 **Genera movimiento de cola relajada** usando funciones sinusoidales para crear ondulaciones suaves. Combina oscilación lateral con flotación vertical suave, incluyendo la preferencia direccional personal del gato. El resultado es un movimiento natural y orgánico que simula un gato descansando pero atento.
- `patron_cola_curiosa()` → Vector3 **Crea movimientos de exploración** más animados con oscilaciones complejas. Utiliza múltiples frecuencias sinusoidales superpuestas para generar patrones complejos pero naturales. Incluye "caprichos" adicionales que simulan la naturaleza impredecible de los gatos curiosos.

Sistema de Estados

BehaviorHandler - Gestión de Comportamientos

El **sistema de estados** implementa una **máquina de estados finitos** que coordina los diferentes comportamientos del gato mediante clases especializadas. Cada comportamiento (caminar, saltar, correr) se encapsula en su propia clase que hereda de `BaseBehavior`, proporcionando **separación clara de responsabilidades** y facilitando la adición de nuevos comportamientos. El sistema utiliza **transiciones**

automáticas basadas en condiciones del entorno como distancia al objetivo, detección de obstáculos, y estado del terreno.

La arquitectura permite **comportamientos complejos emergentes** donde cada estado puede evaluar condiciones específicas y solicitar transiciones a otros estados. El `BehaviorHandler` actúa como **coordinador central** que valida las transiciones y asegura que solo ocurran cambios de estado válidos, previniendo inconsistencias en el comportamiento del gato.

Clase Base BaseBehavior

```
class BaseBehavior:
    var context: Node3D
    func enter(): pass
    func update(): pass
    func exit(): pass
    func can_transition_to(behavior_name: String) -> bool: return false
```

La **clase base** define la interfaz común para todos los comportamientos. `enter()` se ejecuta al activar el comportamiento, permitiendo inicialización específica. `update()` se llama cada frame y contiene la lógica principal del comportamiento, retornando el nombre del próximo estado deseado. `exit()` maneja limpieza al abandonar el comportamiento. `can_transition_to()` define qué transiciones son válidas desde este estado.

WalkBehavior - Comportamiento de Caminar

Implementa la locomoción básica del gato con **detección inteligente de condiciones** para transiciones automáticas. Evalúa continuamente la distancia al objetivo para determinar si debe cambiar a modo correr, detecta obstáculos que requieren salto, y valida que cada paso sea anatómicamente viable. Coordina el **patrón de marcha de cinco estados** (PASO_1 a PASO_5) asegurando que las patas se muevan en secuencia apropiada para mantener estabilidad felina.

El comportamiento implementa **umbrales adaptativos** donde las distancias y condiciones se evalúan dinámicamente. Utiliza las funciones `debe_avanzar()` y `validar_siguiente_paso()` para tomar decisiones inteligentes sobre cuándo mover cada pata y hacia dónde dirigirla.

JumpBehavior - Comportamiento de Salto

Maneja la secuencia completa de salto desde preparación hasta aterrizaje, implementando **fases específicas** con transiciones automáticas. Durante `SALTO_PREP`, el gato ajusta posición de patas para balance óptimo y calcula trayectoria de salto. En `SALTO`, las patas delanteras se extienden hacia el objetivo mientras las traseras proporcionan impulso. Durante `ATERRIZAJE`, las patas traseras alcanzan el objetivo completando la secuencia.

El sistema **detecta automáticamente** si el salto es hacia arriba (obstáculo) o hacia abajo (bajada) ajustando la física y trayectoria apropiadamente. Utiliza la variable `salto_es_bajada` para modificar patrones de movimiento y alturas de arco del salto.

RunBehavior - Comportamiento de Carrera

Implementa carrera de alta velocidad con **saltos extendidos** para cubrir distancias largas rápidamente. Similar al comportamiento de salto pero con zancadas más largas y velocidades aumentadas. Maneja tres fases: `CORRER_PREP` para posicionamiento inicial, `CORRER_SALTO` para zancadas extendidas, y `CORRER_ATERRIZAJE` para completar cada ciclo.

El comportamiento **se mantiene activo** mientras la distancia al objetivo sea mayor que `DISTANCIA_PARAR_CORRER`, creando un **bucle continuo** de zancadas hasta que el gato se acerque suficientemente al objetivo. Incluye detección de `salto_es_alto` para ajustar la trayectoria cuando el terreno requiere mayor elevación.

BehaviorHandler - Coordinador Central

```
class BehaviorHandler:
    var behaviors = {}
    var current_behavior: BaseBehavior
    var current_behavior_name: String = ""
```

El **coordinador central** mantiene un diccionario de todos los comportamientos disponibles y gestiona las transiciones entre ellos. `start_behavior()` activa un comportamiento específico llamando a su método `enter()`. `update()` ejecuta el comportamiento actual y procesa solicitudes de transición. `transition_to()` valida que la transición sea permitida antes de cambiar estados.

El sistema **previene transiciones inválidas** verificando que el comportamiento actual permita la transición solicitada mediante `can_transition_to()`. Esto asegura que el gato no pueda, por ejemplo, saltar mientras ya está en medio de un salto, manteniendo **coherencia comportamental** realista.

Rigging y Animación

targets_cat.gd - Sistema IK

Este archivo implementa el **sistema de Inverse Kinematics (IK)** que permite al gato moverse de forma natural y adaptarse a cualquier terreno. El sistema **no utiliza animaciones pregrabadas**, sino que calcula posiciones de huesos en tiempo real basándose en targets objetivos. Esto permite que el gato se adapte automáticamente a superficies irregulares, escaleras, techos inclinados y cualquier geometría compleja del entorno.

El sistema maneja **rigging procedural completo** incluyendo patas, columna vertebral y cola. Cada componente tiene su propio solver IK que trabaja independientemente pero de forma coordinada. La **columna vertebral** utiliza un sistema de física de resorte para movimientos naturales que responden a la actividad del gato. Los **targets** se posicionan automáticamente basándose en las decisiones del sistema principal de locomoción.

Componentes Principales

Targets IK: Los targets son **objetivos virtuales** que definen dónde deben posicionarse las extremidades. `t_backL/R` controlan las patas traseras y se mueven en coordinación con el patrón de marcha. `t_frontL/R` manejan las patas delanteras con movimiento independiente para exploración y balance. `t_tail` controla la cola integrándose con el sistema emocional de movimiento de cola.

Solvers IK: Los solvers son **algoritmos matemáticos** que calculan las rotaciones de huesos necesarias para alcanzar los targets. `ik_backL/R` resuelven las piernas traseras considerando las limitaciones anatómicas felinas. `ik_frontL/R` manejan las piernas delanteras con mayor flexibilidad para comportamientos exploratorios. `ik_tail` controla la cadena de huesos de la cola para movimientos fluidos y naturales.

Sistema de Columna Vertebral

- `update_column(skeleton, bones, target, delta)` **Actualiza la columna vertebral** usando un sistema de **física de resorte avanzado** que simula la flexibilidad natural de la columna felina. El sistema calcula fuerzas de resorte basándose en la diferencia entre la posición actual y la posición deseada, aplicando amortiguación para movimientos suaves. Utiliza **integración de Verlet** para estabilidad numérica y respuesta natural a cambios de dirección y velocidad.

Parámetros de Resorte:

- `spring_stiffness`: **Rigidez del resorte** (250.0) que determina qué tan rápido la columna responde a cambios
- `spring_damping`: **Amortiguación** (15.0) que previene oscilaciones excesivas y proporciona movimiento natural
- `mass`: **Masa simulada** de la cabeza (1.0) que afecta la inercia del sistema de resorte

Detección de Terreno

El sistema implementa **detección inteligente de terreno** que permite al gato adaptarse automáticamente a cualquier superficie. Utiliza **raycasting direccional** desde posiciones elevadas hacia abajo para encontrar puntos de contacto válidos. El sistema **no solo detecta altura** sino que también valida la viabilidad anatómica de cada paso, asegurando que el gato no intente movimientos imposibles.

- `obtener_punto_mas_alto(x, z, grupo_nombre)` → Vector3 **Realiza detección de terreno** usando raycasting desde una posición elevada hasta encontrar la superficie más alta en las coordenadas especificadas. El sistema filtra por grupos de objetos (como "suelo") para distinguir entre diferentes tipos de superficies. Proporciona fallback seguro retornando coordenadas básicas si no encuentra intersección válida.

- `validar_siguiente_paso()` → bool **Verifica viabilidad anatómica** del próximo paso antes de ejecutarlo. Calcula las posiciones proyectadas de las patas delanteras, evalúa las alturas del terreno en esos puntos, y determina si el gato puede físicamente alcanzar esas posiciones sin comprometer su estabilidad. Incluye búsqueda de rutas alternativas si el camino directo no es viable.

Cámara y Control

Camera.gd - Sistema de Seguimiento

Este archivo implementa un **sistema de cámara dinámico** diseñado específicamente para seguir al gato de manera cinematográfica y funcional. La cámara **no es estática** sino que responde inteligentemente al comportamiento del gato, proporcionando los mejores ángulos para observar la locomoción y exploración. Incluye **efectos visuales avanzados** que mejoran la experiencia de usuario y proporcionan feedback visual para interacciones.

El sistema implementa **múltiples modos de operación** desde seguimiento automático hasta control manual completo. Los **efectos visuales** incluyen explosiones procedurales en puntos de click, cursores 3D que se adaptan al terreno, y partículas dinámicas que responden a la interacción del usuario. La cámara utiliza **interpolación suave** para transiciones naturales y evitar movimientos bruscos que puedan distraer de la observación del comportamiento felino.

Opciones de la Cámara

- **Seguimiento suave**: Utiliza **interpolación linear (lerp)** para seguir al gato con movimientos fluidos y naturales
- **Rotación libre**: Activada con **clic derecho** permite explorar el entorno desde cualquier ángulo manteniendo el enfoque
- **Zoom dinámico**: **Rueda del mouse** controla la distancia con límites configurables para perspectivas óptimas
- **Efectos visuales**: **Sistema de partículas** y explosiones que responden a clicks proporcionando feedback visual inmediato

Configuración Visual

Efectos de Explosión: Los efectos de explosión utilizan **geometría procedural** para crear ondas expansivas realistas. `radio_explosion_max` define el tamaño máximo del efecto visual en unidades del mundo. `duracion_explosion` controla cuánto tiempo dura la animación completa del efecto. `particulas_por_explosion` determina la densidad visual del efecto, balanceando impacto visual con rendimiento.

- `radio_explosion_max`: Radio máximo de ondas que se expande desde el punto de click
- `duracion_explosion`: Duración temporal del efecto completo de explosión
- `particulas_por_explosion`: Cantidad de partículas generadas para densidad visual

GENERACIÓN PROCEDURAL

Sistema de creación dinámica de ciudades y arquitectura que genera entornos urbanos complejos inspirados en la arquitectura andina, específicamente Cusco, utilizando algoritmos matemáticos avanzados para crear espacios coherentes y navegables.

Generador Principal

CityGeneratorV3.gd - Coordinador Central

Este archivo es el **core del sistema de generación** que coordina todos los subsistemas para crear ciudades procedurales coherentes. Implementa un **sistema de chunks dinámico** que genera contenido basándose en la posición del jugador, optimizando rendimiento y memoria. El sistema utiliza **algoritmos de Voronoi** para crear divisiones orgánicas del espacio urbano que simulan el crecimiento natural de ciudades históricas.

La generación sigue un **enfoque jerárquico** donde primero se establecen las estructuras principales (avenidas y distritos) y luego se refinan los detalles (calles, manzanas, edificios). El sistema **mantiene coherencia espacial** asegurando que los elementos generados en diferentes momentos se integren perfectamente. Incluye **sistema de debug visual** que permite inspeccionar el proceso de generación en tiempo real.

Configuración de Chunks

El sistema utiliza **dos niveles de chunks** para optimizar tanto el rendimiento como la densidad de detalles:

Chunks de Avenidas: Manejan la **estructura principal de la ciudad** con elementos de gran escala. `avenue_chunk_size` define chunks de 1300 unidades que contienen las arterias principales y definen la estructura básica urbana. `avenue_render_distance` de 3 chunks asegura contexto visual suficiente sin comprometer rendimiento. `max_avenue_points_per_chunk` limita a 1 punto por chunk para mantener escala urbana apropiada.

Chunks de Calles: Gestionan **detalles de nivel medio** como calles secundarias y conexiones locales. `street_chunk_size` de 200 unidades proporciona resolución apropiada para navegación del gato. `street_render_distance` de 4 chunks mantiene suficiente detalle para movimiento fluido. `max_street_points_per_chunk` de 1 punto asegura densidad apropiada sin sobrecarga computacional.

Generación de Terreno

Configuración de Terreno: El sistema genera **terreno procedural** que se integra con la arquitectura urbana. `terrain_resolution` de 128 vertices proporciona detalle suficiente para navegación mientras mantiene rendimiento. `terrain_material` aplica materiales PBR para realismo visual. `generate_terrain` permite activar/desactivar la generación según necesidades de rendimiento.

SpatialChunkManager.gd - Sistema de Chunks

Este archivo implementa la **lógica de gestión espacial** que divide el mundo en chunks manejables y coordina su generación/descarga dinámica. Utiliza **algoritmos de hash determinístico** para asegurar que el mismo contenido se genere siempre en las mismas coordenadas. El sistema mantiene **coherencia temporal** asegurando que los chunks se generen en el orden correcto para evitar dependencias rotas.

La gestión incluye **optimización de memoria** descargando chunks lejanos y manteniendo cache de elementos frecuentemente accedidos. Implementa **sistema de prioridades** donde chunks más cercanos al jugador se procesan primero. El sistema también maneja **asignación de puntos a distritos** usando geometría computacional para determinar qué puntos pertenecen a qué áreas urbanas.

Funciones de Coordenadas

- `get_avenue_chunk_coord(world_x, world_y)` → Vector2i **Convierte coordenadas del mundo** a coordenadas de chunk de avenida usando división entera. Esto permite **mapeo determinístico** donde las mismas coordenadas mundiales siempre resultan en el mismo chunk, asegurando consistencia en la generación procedural.
- `get_street_chunk_coord(world_x, world_y)` → Vector2i
  Convierte coordenadas del mundo a coordenadas de chunk de calle con resolución más fina. Utiliza el mismo principio determinístico pero con divisiones más pequeñas para mayor detalle en la navegación local.

Generación de Puntos

- `get_avenue_points_in_render_distance(player_pos)` → Array **Obtiene puntos de avenida** en el área de renderizado actual. Calcula el chunk central basándose en la posición del jugador, itera sobre todos los chunks vecinos en el rango de distancia, y genera/obtiene los puntos de cada chunk usando algoritmos de hash determinístico.

- `get_street_points_in_render_distance(player_pos)` → Array **Obtiene puntos de calle** con mayor densidad para navegación detallada. Utiliza el mismo algoritmo de área pero con chunks más pequeños, proporcionando suficientes puntos para generar calles locales y conexiones entre edificios.

Asignación de Distritos

- `assign_street_points_to_districts(districts, street_points)` → Array **Asigna puntos a distritos** usando **geometría computacional** avanzada. Para cada punto de calle, determina qué distrito lo contiene usando algoritmos de "punto-en-polígono". Mantiene estructuras de datos eficientes para consultas rápidas y actualiza asignaciones dinámicamente cuando se generan nuevos distritos.

Gestión de Bloques

ManzanaManager.gd - Manager de Manzanas

Este archivo maneja la **generación y administración de manzanas urbanas** (bloques de edificios), implementando un sistema de estabilidad que asegura que los bloques solo se generen cuando tienen suficiente contexto geométrico. Utiliza **verificación de estabilidad** para prevenir generación prematura que podría resultar en geometría inconsistente o conectividad rota.

El manager implementa **generación bajo demanda** donde las manzanas se crean solo cuando el jugador está suficientemente cerca y todos los chunks necesarios están cargados. Mantiene **registro de manzanas generadas** para evitar duplicación. El sistema también coordina con el generador de edificios para poblar las manzanas con arquitectura apropiada.

Registro de Manzanas

- `generate_manzana_if_stable(punto, manzana_polygon)` → bool **Genera manzana condicionalmente** verificando primero que el punto sea geométricamente estable. Esto significa que todos los chunks vecinos están cargados y tienen suficientes puntos para establecer contexto completo. Solo procede con la generación si se cumplen todas las condiciones de estabilidad.

- `is_point_stable(punto)` → bool
  Verifica estabilidad geométrica consultando el StabilityChecker para determinar si un punto tiene suficiente contexto para generar geometría válida. Esto previene artefactos visuales y garantiza coherencia espacial en la ciudad generada.

Creación de Mallas

- `create_manzana_mesh(punto, manzana_polygon)` → Array **Genera geometría 3D** para edificios dentro de la manzana. Primero aplica padding al polígono de la manzana para crear separación entre edificios, luego subdivide el área en lotes individuales, y finalmente genera edificios para cada lote usando el sistema de construcción de casas.

- `subdivide_polygon_into_lots(polygon)` → Array **Subdivide manzanas en lotes** utilizando algoritmos avanzados de subdivisión de polígonos. Aplica simplificación geométrica para eliminar complejidad innecesaria, luego usa el PolygonSubdivider para crear lotes con proporciones apropiadas para construcción de edificios.

StabilityChecker.gd - Validador Geométrico

Este archivo implementa un **sistema de verificación de estabilidad** que determina cuándo es seguro generar geometría procedural. Utiliza **cache temporal** para optimizar verificaciones repetidas y mantiene consistencia en las decisiones de generación. El sistema previene **artefactos geométricos** que pueden ocurrir cuando se genera contenido sin suficiente contexto espacial.

La verificación de estabilidad es **crítica para la calidad** visual y la coherencia espacial. Asegura que cada elemento generado tenga suficiente información de sus vecinos para crear conexiones apropiadas y evitar geometría flotante o desconectada. El sistema implementa **estrategias de cache** para reducir la carga computacional de verificaciones frecuentes.

Sistema de Caché

- `is_street_point_stable(punto, chunk_manager)` → bool **Verifica estabilidad de puntos de calle** asegurando que todos los 8 chunks vecinos estén cargados y tengan puntos válidos. Utiliza cache temporal para evitar recálculos innecesarios. Solo retorna verdadero cuando el punto tiene contexto geométrico completo para generar conexiones de calle válidas.
- `is_avenue_point_stable(punto, chunk_manager)` → bool **Verifica estabilidad de puntos de avenida** usando los mismos criterios pero para chunks de avenida de mayor escala. Asegura que las arterias principales se generen con continuidad apropiada y conexiones coherentes con la estructura urbana existente.

Funciones de Debug

- `debug_point_stability(punto, chunk_manager, chunk_type)` → Dictionary **Proporciona información detallada** sobre el estado de estabilidad de cualquier punto. Incluye información sobre chunks vecinos cargados, cantidad de puntos en cada chunk, y razones específicas por las que un punto puede no ser estable. Esencial para debugging del sistema de generación.

Utilidades Geométricas

VoronoiGenerator.gd - Diagramas de Voronoi

Este archivo implementa **generación de diagramas de Voronoi** para crear divisiones orgánicas del espacio urbano que simulan el crecimiento natural de ciudades. Utiliza **triangulación de Delaunay** como base matemática y luego calcula los polígonos duales de Voronoi. El sistema maneja **recorte por límites** para asegurar que las regiones generadas se ajusten a áreas específicas de la ciudad.

Los diagramas de Voronoi son **fundamentales para la naturalidad** de la ciudad generada, ya que crean divisiones irregulares pero equilibradas que se asemejan a la manera en que las ciudades reales crecen orgánicamente. El sistema implementa **optimizaciones específicas** para casos de uso urbano, incluyendo manejo de casos límite y validación de geometría resultante.

Generación Principal

- `generate_voronoi(sites, bounds)` → Array **Genera diagrama de Voronoi completo** para un conjunto de puntos semilla dentro de límites específicos. Utiliza el addon de triangulación de Delaunay para crear la estructura base, calcula circuncentros de triángulos para determinar vértices de Voronoi, y recorta los polígonos resultantes por los límites del área de interés.

- `generate_voronoi_for_district(sites, district_polygon)` → Dictionary
  Genera Voronoi especializado recortado por un polígono de distrito específico. Esto permite crear subdivisiones orgánicas dentro de áreas urbanas predefinidas, manteniendo coherencia con la estructura de distrito mientras proporcionando variedad interna.

Funciones Auxiliares

- `get_circumcenter(p1, p2, p3)` → Vector2 **Calcula el circuncentro** de un triángulo, que es el punto equidistante de los tres vértices. Esta es la base matemática para generar los vértices de los polígonos de Voronoi. Incluye manejo de casos degenerados donde los puntos son colineales.

PolygonSubdivider.gd - Subdivisión de Polígonos

Este archivo implementa **algoritmos avanzados de subdivisión** de polígonos para dividir manzanas urbanas en lotes apropiados para construcción. El sistema utiliza **múltiples estrategias** de subdivisión y selecciona automáticamente la mejor basándose en la forma del polígono de entrada. Implementa **validación de proporciones** para asegurar que los lotes resultantes sean apropiados para construcción realista.

El sistema es **crítico para la calidad** de la generación urbana, ya que determina cómo se organizan los edificios dentro de cada manzana. Utiliza **constantes configurables** para controlar aspectos como ratio de aspecto máximo y ancho mínimo de lotes. Implementa **estrategias de fallback** para manejar polígonos complejos o irregulares que no se subdividen fácilmente.

Algoritmos de Subdivisión

- `subdivide_polygon(polygon, lot_size)` → Array **Función principal de subdivisión** que analiza el polígono de entrada y selecciona la estrategia óptima de subdivisión. Calcula el área objetivo para cada lote, determina el número apropiado de subdivisiones, y prueba múltiples algoritmos para encontrar el mejor resultado que respete las restricciones de proporción.

Estrategias de Subdivisión

- `try_smart_grid_subdivision(polygon, num_lots, target_area)` → Array **Intenta subdivisión en grilla inteligente** que respeta la forma del polígono original. Calcula dimensiones óptimas de grilla basándose en el polígono envolvente y el número deseado de lotes, verificando que cada celda resultante tenga proporciones apropiadas antes de proceder.

- `try_adaptive_strip_subdivision(polygon, num_lots, target_area)` → Array **Implementa subdivisión en franjas** que se adapta a polígonos alargados. Determina la dirección óptima para las franjas basándose en las proporciones del polígono, y puede subdividir recursivamente si las franjas resultantes son demasiado alargadas.

- `try_organic_subdivision(polygon, num_lots, target_area)` → Array **Crea subdivisiones orgánicas** usando una aproximación similar a Voronoi. Genera puntos semilla bien distribuidos dentro del polígono y crea regiones naturales alrededor de cada punto, resultando en lotes con formas más irregulares pero naturales.

Validación de Geometría

- `has_good_aspect_ratio(polygon)` → bool **Verifica proporciones apropiadas** calculando el ratio de aspecto del bounding box del polígono. Asegura que los lotes no sean excesivamente alargados o delgados, lo que resultaría en edificios poco realistas.

Constantes de Validación:

- `MAX_ASPECT_RATIO`: 2.5 - Limita el ratio máximo ancho/alto para prevenir lotes excesivamente alargados
- `MIN_LOT_WIDTH`: 15.0 - Asegura ancho mínimo absoluto para permitir construcción de edificios apropiados

GeometryUtils.gd - Utilidades de Geometría

Este archivo proporciona **funciones geométricas fundamentales** utilizadas por todo el sistema de generación procedural. Implementa **algoritmos computacionales** robustos para operaciones complejas como recorte de polígonos, detección de inclusión punto-en-polígono, y manipulación de geometría 2D. Estas utilidades son la **base matemática** sobre la que se construyen los sistemas de generación más complejos.

Las funciones están optimizadas para **uso intensivo** ya que se llaman frecuentemente durante la generación de ciudad. Implementan algoritmos clásicos de geometría computacional con **manejo robusto de casos límite** y validación de entrada para prevenir errores en cálculos geométricos críticos.

Operaciones con Polígonos

- `point_in_polygon(point, polygon)` → bool **Implementa test de inclusión** usando el algoritmo de ray casting para determinar si un punto está dentro de un polígono arbitrario. Lanza un rayo desde el punto hacia el infinito y cuenta las intersecciones con los bordes del polígono. Un número impar de intersecciones indica que el punto está dentro.

- `clip_polygon(subject, clip)` → Array **Recorta polígonos** usando el algoritmo de Sutherland-Hodgman, que corta iterativamente el polígono sujeto contra cada borde del polígono de recorte. Es fundamental para crear lotes que se ajusten exactamente a los límites de manzanas y distritos.

- `get_polygon_bounds(vertices)` → Rect2 **Calcula bounding box** encontrando las coordenadas mínimas y máximas en X e Y de todos los vértices. Utilizado para optimizaciones espaciales y cálculos de área aproximada.

- `create_padded_polygon(vertices, padding)` → Array **Crea polígono con separación interna** reduciendo uniformemente el tamaño del polígono hacia su centro. Esto es esencial para crear separación entre edificios dentro de manzanas, evitando que se toquen directamente.

CONSTRUCCIÓN DE EDIFICIOS

Sistema de generación procedural de arquitectura que crea edificios únicos con detalles realistas, incluyendo diferentes tipos de techos, sistemas de ventanas, y adaptación automática al terreno usando algoritmos de ruido procedural.

Constructor Principal

HouseBuilder.gd - Generador de Edificios

Este archivo es el **corazón del sistema arquitectónico** que genera casas procedurales únicas adaptadas al estilo arquitectónico de Cusco. Implementa **múltiples tipos de techo** (a dos aguas, una agua, piramidal) con **generación automática de buhardillas y ventanas**. Se utilizan **números aleatorios determinísticos** basados en posición para asegurar que el mismo edificio se genere siempre en la misma ubicación.

La generación incluye **detalles** como aleros, marcos de ventanas, diferentes materiales para techos y paredes, y **adaptación automática al terreno** usando ruido Perlin.

Configuración de Materiales

Materiales Predefinidos: El sistema utiliza **materiales PBR realistas** cargados como recursos. `roof_material` aplica texturas de tejas con mapas de normal y paralaje para techos realistas. `wall_material` usa texturas de piedra o adobe apropiadas para arquitectura andina. `glass_material` y `frame_material`.

Configuración de Edificios

- `config(base_quad, square_threshold)` → Dictionary **Genera configuración procedural** para cada edificio basándose en su polígono base. Utiliza números aleatorios determinísticos para calcular altura total, altura de techo, tipo de techo (según número de lados del polígono), cantidad de buhardillas, y proporciones específicas. Los parámetros se escalan apropiadamente para mantener proporciones arquitectónicas realistas.

Parámetros Generados:

- `height`: **Altura total** calculada como múltiplo de altura por piso (2.3-2.6m) por número aleatorio de pisos
- `height_roof`: **Altura del techo** como porcentaje (20-32%) de la altura total del edificio
- `roof_type`: **Tipo de techo** seleccionado basándose en la complejidad del polígono base
- `num_dormers`: **Cantidad de buhardillas** determinada aleatoriamente para techos a dos aguas
- `percentage`: **Proporción de ancho** de buhardillas relativo al ancho total del techo

Tipos de Techo

El sistema implementa **cuatro tipos principales de techo** cada uno con sus propias características y casos de uso específicos:

Techo a Dos Aguas (Tipo 0)

```
roof_gabled(top_quad, height_roof, vertices, faces, eave, num_dormers, percentage)
```

Implementa techos tradicionales con dos superficies inclinadas que se encuentran en una cresta central. Calcula automáticamente la orientación óptima basándose en las proporciones del edificio, determina las posiciones de las crestas usando intersección de planos, y genera aleros proceduralmente para mayor realismo. Incluye soporte para buhardillas con ventanas integradas.

Techo Plano (Tipo 1)

Genera techos planos El más simplista. Simplemente crea un plano elevado sobre el polígono base.

Techo a Una Agua (Tipo 2)

```
roof_one_slope(top_quad, height_roof, vertices, faces, eave)
```

Crea techos inclinados en una sola dirección, apropiados para edificios adosados o con orientación específica. Calcula la inclinación y genera aleros en tres lados manteniendo un lado.

Techo Piramidal (Tipo 3)

```
roof_pyramid(top_quad, height_roof, vertices, faces, eave)
```

Genera techos piramidales para polígonos irregulares o edificios especiales. Calcula el centroide del polígono superior, crea un pico central, y genera superficies triangulares desde cada lado hacia el pico central.

Sistema de Ventanas

Generación de Ventanas

El sistema de ventanas es implementado con múltiples componentes y adaptación automática a diferentes tipos de superficies arquitectónicas.

- `create_window(width, height, center, normal)` → Node3D **Crea ventanas complejas** con múltiples componentes realistas. Genera un sistema de coordenadas local basado en la normal de la superficie, crea marco exterior de cuatro piezas, añade separador central vertical para ventanas de doble hoja, y coloca cristales transparentes con materiales apropiados. Cada ventana es un nodo completo con geometría 3D y materiales aplicados.

Componentes de Ventana:

- **Marco exterior**: Cuatro piezas de madera o metal que forman el perímetro de la ventana
- **Separador central vertical**: Divide la ventana en dos hojas como es tradicional en arquitectura andina
- **Cristales**: Dos paneles transparentes con materiales que incluyen reflexión y transparencia

Ventanas en Buhardillas

- `generate_dormer_with_window(...)` → Node3D **Genera buhardillas arquitectónicas** con ventanas integradas en techos a dos aguas. Calcula la intersección del plano del techo con la proyección vertical de la buhardilla, crea la geometría del dormer incluyendo paredes laterales y frontal, y coloca una ventana apropiadamente dimensionada en la cara frontal. Las buhardillas añaden variedad visual y funcionalidad arquitectónica.

- `create_window_on_polygon(polygon_vertices, width_ratio, height_ratio)` → Node3D **Adapta ventanas a polígonos irregulares** como triángulos en frontones de techos. Calcula el centro del polígono, determina la normal de la superficie, proyecta el polígono a un sistema de coordenadas 2D local, calcula dimensiones apropiadas basándose en los ratios proporcionados, y crea una ventana que se ajusta visualmente a la superficie disponible.

Generación de Terreno

Sistema de Ruido

El sistema implementa **generación de terreno procedural** que se integra perfectamente con la arquitectura urbana, asegurando que los edificios se asienten naturalmente en el paisaje.

- `generate_y_offset_from_noise(base_poly)` → float **Genera elevación de terreno** usando ruido Perlin para crear variaciones naturales de altura. Evalúa el ruido en el centro y todas las esquinas del polígono del edificio, toma el valor mínimo para asegurar que todo el edificio se asiente correctamente, y mapea el resultado a un rango de elevación apropiado. Esto previene edificios flotantes y crea integración natural con el terreno.

Configuración de Ruido:

- `noise_type`: **TYPE_PERLIN** para patrones naturales de elevación del terreno
- `seed`: **54321** - semilla fija para generación determinística
- `frequency`: **0.005** - frecuencia baja para transiciones suaves entre elevaciones
- `min_offset`: **-50.0** / `max_offset`: **50.0** - rango de variación de altura en unidades del mundo

Función Principal

- `build_house_with_all_windows(base_poly, door_index)` → Dictionary **Función principal de construcción** que coordina todos los subsistemas para crear un edificio completo. Aplica correcciones al polígono base, genera configuración procedural, aplica offset de terreno a todos los vértices, crea polígono superior elevado, genera paredes laterales conectando base y techo, añade piso interior, selecciona y ejecuta tipo de techo apropiado, y finalmente genera todas las ventanas incluyendo las de buhardillas.

Retorna:

- `house`: **MeshInstance3D** completo del edificio con todas las superficies y materiales aplicados
- `windows`: **Array de nodos** conteniendo todas las ventanas generadas como objetos 3D separados

Modos de DEBUG

El sistema incluye **herramientas de debug visual** integradas que permiten inspeccionar el proceso de generación en tiempo real y diagnosticar problemas de rendimiento o geometría. El sistema de debug opera mediante **overlay gráfico 2D** que se superpone a la vista 3D, convirtiendo coordenadas del mundo 3D a coordenadas de pantalla para visualización.

Debug de Generación Procedural

Visualización de Chunks

El sistema de debug de chunks implementa **proyección 3D a 2D** para dibujar rectángulos que representan los límites de cada chunk en el mundo. Utiliza la función `cam.unproject_position()` para convertir las esquinas de cada chunk desde coordenadas del mundo 3D a píxeles de pantalla, permitiendo visualizar exactamente qué áreas están siendo procesadas por el sistema de generación.

La **visualización diferenciada por colores** permite distinguir inmediatamente entre tipos de chunks: **azul para chunks de avenidas** (estructuras principales de gran escala), **verde para chunks de calles** (detalles locales de navegación), y **superposición de ambos** para ver la relación jerárquica entre sistemas. Cada chunk se dibuja como un **rectángulo con bordes sólidos** que representa exactamente el área de influencia de ese chunk en el mundo.

Debug de Distritos y Manzanas

El sistema incluye **visualización de polígonos complejos** para distritos (cian) y manzanas. Los **distritos** se dibujan como polígonos irregulares que muestran las divisiones de Voronoi, permitiendo verificar que las divisiones urbanas se están generando correctamente. Las **manzanas** se visualizan con diferentes colores según su estado: **verde lima para manzanas generadas** (con edificios construidos), **naranja para manzanas pendientes** (esperando estabilidad), y **amarillo para manzanas visibles** pero no procesadas.

Debug de Estabilidad

Las herramientas de debug incluyen **visualización de estabilidad de puntos** que utiliza un **sistema de colores semafórico** para mostrar el estado de cada punto en el sistema de generación. Los **puntos rojos** indican chunks que aún no están cargados o no tienen suficientes vecinos. Los **puntos amarillos** muestran chunks parcialmente estables. Los **puntos verdes** representan chunks completamente estables listos para generación.

Esta visualización es **crucial para diagnosticar** problemas como edificios que no aparecen (falta de estabilidad), geometría fragmentada (chunks cargándose en orden incorrecto), o problemas de rendimiento (demasiados chunks inestables creando trabajo innecesario).

Información de Estadísticas

El overlay incluye **panel de información en tiempo real** que muestra:

- **Conteos de chunks** cargados por tipo (avenidas/calles)
- **Cantidad de puntos** generados en cada sistema
- **Manzanas estables vs pendientes** para monitorear progreso de generación
- **Posición del jugador** en coordenadas del mundo
- **Instrucciones de control** para las teclas de debug

Debug del Sistema Felino

Visualización de Targets IK

El sistema de debug del gato implementa **esferas de debug 3D** que se posicionan en el mundo para mostrar visualmente el estado interno del sistema de locomoción. Cada **target de pata** se representa con una esfera verde que muestra exactamente dónde el sistema IK está intentando posicionar cada extremidad. Las **posiciones actuales de patas** se muestran con esferas azules, permitiendo ver la diferencia entre objetivo y realidad.

La **visualización de objetivos** incluye una esfera roja para el `punto_objetivo` que muestra hacia dónde se dirige el gato, y una esfera dorada para la `cola_tip` que representa la posición actual de la cola. Esto permite **debugging visual inmediato** de problemas como patas que no alcanzan sus objetivos, cola que no sigue al cuerpo correctamente, o objetivos mal posicionados.

Controles de Debug

Sistema de Activación

- **F2**: Activa/desactiva el **overlay completo de debug visual**, alternando entre vista limpia y vista con información técnica superpuesta
- **F3**: **Alterna entre modos de visualización** con tres estados: solo chunks de avenidas, solo chunks de calles, o ambos tipos simultáneamente
- **Vista superior automática**: El sistema **detecta automáticamente** cuando la cámara está en ángulo cenital (menos de 42° desde vertical) y activa visualizaciones apropiadas para vista de planta

Fallback de Controles

El sistema implementa **detección de teclas de fallback** para asegurar compatibilidad en diferentes configuraciones. Si las acciones de input mapeadas no funcionan, el sistema detecta directamente las teclas F2 y F3 usando `Input.is_key_pressed()` con prevención de activación repetida mediante metadata temporal.

Información Contextual

El overlay de debug incluye **ayuda contextual** que muestra automáticamente qué controles están disponibles y qué representan los diferentes colores y símbolos en pantalla. Esta información se actualiza dinámicamente según el modo de debug activo, proporcionando **guía inmediata** sin necesidad de consultar documentación externa.

NOTAS ADICIONALES

Limitaciones Conocidas

- El **sistema IK** puede presentar glitches menores en terrenos extremadamente irregulares con pendientes superiores a 60 grados
- La **generación procedural** está optimizada para ciudades de tamaño medio (aproximadamente 100 manzanas simultáneas)

- Algunos **materiales con Parallax Mapping** pueden no renderizar correctamente en plataformas móviles o hardware más antiguo debido a limitaciones en el cálculo de tangentes

Trabajo Futuro

Características Planificadas

- **Sistema de clima dinámico** que afecte el comportamiento del gato (refugiarse de lluvia, buscar sombra)
- **Múltiples gatos** con interacción social y comportamientos de manada
- **Generación de vegetación** procedural integrada con la arquitectura urbana
- **Ciclos día/noche** con comportamiento adaptativo del gato según iluminación

Mejoras de Rendimiento

- **Implementación de GPU compute shaders** para generación de terreno y geometría compleja
- **Optimización de algoritmos geométricos** usando estructuras de datos espaciales más eficientes
- **Sistema de streaming de assets** para reducir uso de memoria en ciudades extensas

> Consejo: Revisa regularmente las actualizaciones del proyecto para nuevas características y optimizaciones.

Créditos

Desarrolladores Principales:

- **Cesar Aaron Perales Rosales** - Arquitectura del sistema de locomoción felina, implementación de máquinas de estado, algoritmos de generación de lotes urbanos, y coordinación general del proyecto
- **Melisa Karen Rivera Alagon** - Sistema complejo de Inverse Kinematics para columna vertebral, implementación de física de resorte, generación procedural de edificios con sistemas de tejas y detalles arquitectónicos
- **Fabryzzio Josue Meza Torres** - Desarrollo de materiales PBR avanzados, implementación de sistemas de cielo dinámico, redacción y estructuración de documentación técnica completa
- **Flavio Tipula** - Diseño e implementación de interfaces de usuario en Godot, integración de sistemas de menús, y optimización de flujos de trabajo del editor

> Nota: El proyecto acepta contribuciones de la comunidad, aunque el equipo principal no garantiza respuesta inmediata a pull requests debido a compromisos académicos actuales.