# Tunea Tu Combi: construcción y simulación física de vehículos 2D

Kiara Balcazar, Juan Diego Castro, Kayla De Vivanco y Alejandro Ore

***Resumen*—*Tunea Tu Combi* es un videojuego 2D en el que el jugador construye una combi sobre una cuadrícula y la prueba en terrenos irregulares hasta alcanzar una meta. El proyecto toma como referencia conceptual el ciclo de construcción, simulación y reintento de juegos como *Bad Piggies*, pero implementa su propio modelo de piezas, motor de *rigid bodies* (cuerpos rígidos), editor de terrenos y sistema de evaluación. La implementación, desarrollada en C++23 con raylib, convierte una configuración discreta de piezas en *rigid bodies* conectados por *joints*. El mundo resuelve detección de colisiones, impulsos secuenciales, fricción, corrección posicional y rotura estructural. Los terrenos se validan, se muestrean mediante curvas Catmull–Rom, se triangulan y se reutilizan para renderizado y colisión. Se presentan la arquitectura, los fundamentos de los algoritmos, la integración con gameplay, las pruebas geométricas disponibles y los pasos para reproducir el proyecto en sus versiones nativa y web.**

***Index Terms*—computación gráfica, *rigid bodies*, colisiones 2D, Catmull–Rom, triangulación, videojuegos, raylib, WebAssembly.**

## I. Introducción

### A. Del juego recreativo al problema técnico

Una partida del juego *Bad Piggies* puede resumirse como un ciclo sencillo: se observa un nivel, se ensamblan piezas, se inicia una simulación física, se intenta llegar a una meta y se modifica el diseño cuando el vehículo falla. La simplicidad de esta descripción oculta una dificultad de implementación importante. Para que el jugador pueda razonar a partir de cada intento, las piezas deben conservar sus conexiones, el terreno debe ser consistente con lo que se dibuja y la respuesta a los impactos debe ser suficientemente estable para producir resultados comprensibles.

*Tunea Tu Combi* adopta ese ciclo como referencia de diseño, sin intentar reproducir el contenido, los personajes ni la implementación del juego comercial. En este proyecto, el vehículo es una combi que construyes y luego manejas. El conductor debe llegar a la zona de meta, sostener la condición de victoria, y recoger y transportar pasajeros. La Tabla I resume la relación entre el referente conceptual y el producto desarrollado.

Tabla I: Comparación del ciclo de juego de referencia y el proyecto.

| Referente conceptual | *Tunea Tu Combi* |
|---|---|
| *Construcción:* Vehículo modular | *Construcción:* Combi sobre una grilla |
| *Objetivo:* Llegar a una meta | Objetivo: Conductor, pasajeros y ventana de meta |
| Simulación: Piezas conectadas | Simulación: *Rigid bodies* y joints propios |
| Terreno: Obstáculos y desniveles | Terreno: Segmentos lineales y Catmull–Rom |
| Iteración: Fallar y reconstruir | Iteración: Reintentar y optimizar el puntaje |
| Contenido: Niveles diseñados | Contenido: Niveles oficiales y editor personalizado |

El carácter recreativo del producto no reduce su valor técnico. El gameplay depende de problemas centrales de Computación Gráfica: transformaciones entre ‘world space‘ y ‘local space‘, representación de mallas, interpolación de curvas, detección geométrica, dinámica de *rigid bodies*, resolución iterativa de restricciones, renderizado 2D y sincronización entre simulación e interfaz. Por ello, el juego funciona como una aplicación concreta que obliga a integrar estos componentes en lugar de tratarlos como algoritmos aislados.

### B. Objetivo y contribuciones

El objetivo es desarrollar un juego 2D reproducible en el que una construcción discreta pueda transformarse en una estructura física continua y ser evaluada dentro de un nivel. Las principales contribuciones son:

- un editor de construcción basado en celdas, piezas, conectores y reglas de compatibilidad;
- un motor propio de *rigid bodies* con formas geométricas, contactos, fricción, *joints* y subpasos;
- un pipeline de terreno que valida polígonos, muestrea curvas y comparte la geometría entre dibujo y colisión;
- un análisis de conectividad y carga para decidir la rotura local de joints;
- un sistema de niveles, metas, puntuación, grabaciones, ranking y editor de contenido;
- compilación nativa y una ruta de compilación a WebAssembly mediante Emscripten.

El resto del artículo desmonta el producto acabado: primero se presenta la experiencia del jugador, luego la arquitectura que la sostiene, después la geometría y la física, y finalmente las pruebas y la reproducción.



(a) Fase de construcción.



(b) Fase de simulación.




(c) Pantalla de resultados.

Figura 1: Proceso completo del sistema de juego.

## II. El producto y su ciclo de juego

### A. *Experiencia completa*

El bucle principal se muestra en la Fig. 2. El usuario selecciona un nivel, modifica un `BuildModel` sobre un `EditorGrid`, inicia la partida y observa la respuesta de su diseño. La sesión de construcción conserva la información discreta necesaria para editar; la sesión de juego crea una copia de ejecución y la entrega al ensamblador físico. El resultado de la simulación vuelve a la capa de juego como éxito, fallo, rotura, entrega de pasajeros y puntuación.

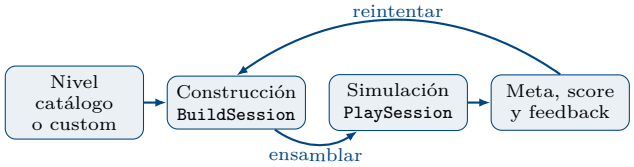



Figura 2: Ciclo de construcción, simulación y reintento.

Las piezas disponibles incluyen cabina del conductor, bloques de pasajeros, ruedas, ventilador, vela y TNT. Una pieza no es solamente una imagen: su definición contiene masa, forma, conectores, orientación, comportamiento y, cuando corresponde, una activación. La combinación de masa, posición, conectividad y fuerzas determina si el diseño consigue avanzar o se desarma.

Tabla II: Piezas y responsabilidades dentro de una construcción.

| Pieza o contenido | Papel en el sistema |
|---|---|
| Cabina del conductor | Aloja al conductor y participa en la condición de meta |
| Bloque de pasajero | Transporta un pasajero asociado |
| Rueda | Contacto circular y movimiento mediante revolución |
| Ventilador | Activación que aporta propulsión |
| Vela | Activación que aplica otro comportamiento de propulsión |
| TNT | Evento explosivo, impulsos y posible rotura en cadena |
| Conectores | Determinan compatibilidad y creación de joints |

Desde la perspectiva del usuario, las operaciones de construcción son colocar, quitar, rotar, seleccionar y reposicionar. La cámara puede moverse con teclado o arrastre y acercarse con la rueda. La simulación se inicia cuando la construcción pasa las reglas mínimas, y las teclas o controles de activación se interpretan mediante `ActivationRegistry`. La interfaz muestra estados de edición y de juego, pero las reglas de compatibilidad y la masa de cada pieza viven en el modelo, no en los botones.




Figura 3: Editor de construcción y representación discreta del vehículo en la grilla.

### B. *Meta, pasajeros y puntuación*

La llegada no se determina únicamente comprobando la posición actual. `GoalWindowTracker` conserva muestras del frame anterior y del actual; así puede detectar una entrada que ocurre entre ambos frames. Cuando el conductor o la parte que lo transporta entra en el área objetivo, comienza una ventana de tres segundos. Una vez iniciada, la ventana no se cancela si el conductor abandona temporalmente el área. Las piezas iniciales que tocan la meta se registran como entregadas y los pasajeros alojados en ellas se incorporan al resultado.

La puntuación se calcula a partir de estadísticas del intento. La implementación asigna 500 puntos por pasajero recogido, 1000 por pasajero entregado, hasta 3000 puntos proporcionales por piezas entregadas y un componente de velocidad de hasta 3000 puntos cuando el tiempo del conductor es menor que 30 segundos. Finalmente, el puntaje ganado se ajusta según la cantidad de piezas iniciales, de modo que un diseño excesivamente grande recibe una penalización de construcción. Estas reglas se implementan en `ScoreRules.cpp` y se comparten con el backend mediante vectores de prueba.

**Algoritmo 1** Actualización de la ventana de meta.

**Require:** Muestras anterior y actual del conductor y área objetivo
**if** el conductor o su vehículo cruza el área **then**
  **if** la ventana no estaba activa **then**
    Calcular la fracción de entrada entre ambos frames
    Iniciar el temporizador de la ventana
  **end if**
  Registrar piezas iniciales y pasajeros entregados
**end if**
**if** tiempo actual − tiempo de entrada ≥ 3000 ms **then**
  **return** Victoria completada
**end if**
**return** Ventana activa o pendiente

El desglose de puntuación puede expresarse como

$$S_{earned} = S_{pickup} + S_{delivery} + S_{parts} + S_{speed}, \quad (1)$$

donde los componentes se obtienen de las estadísticas del 'run'. El ajuste por tamaño de construcción implementado en `ScoreRules.cpp` es:

$$S_{total} = \text{round}\left(\frac{S_{earned}\, M}{M + \max(0, N_{initial} - 2)}\right), \; M = 25. \quad (2)$$

Aquí $N_{initial}$ es la cantidad inicial de piezas. Las reglas rechazan estadísticas negativas, cantidades entregadas mayores que las recogidas y cantidades de piezas entregadas mayores que las iniciales antes de marcar el resultado como válido.

## III. Arquitectura: desmontando el gameplay

### A. Capas y responsabilidades

El código se organiza en módulos con responsabilidades relativamente separadas. La aplicación y sus escenas manejan el ciclo de interfaz; `app/game` contiene niveles, piezas, construcción, sesiones, metas y grabaciones; `math` concentra vectores, matrices, intervalos, AABB, splines y terrenos; `physics` implementa la conversión a cuerpos y el motor de simulación; `rendering` dibuja el mundo, las piezas, los pasajeros y los efectos; y `api`, `infra` y `src/web` cubren la persistencia y el despliegue.

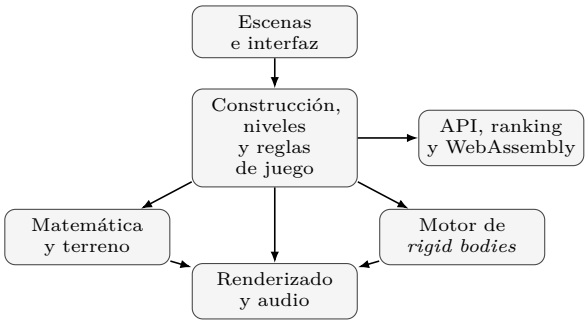



Figura 4: Vista simplificada de la arquitectura.

La transición crítica ocurre en `PhysicsAssembler`. La construcción mantiene celdas, orientaciones y conectores; el ensamblador crea un *rigid body* por pieza, asigna una forma de colisión, conecta cuerpos compatibles y añade el terreno al mundo. El renderizado lee el estado resultante, pero no decide la física. Esta separación evita que una posición dibujada sustituya accidentalmente al estado simulado.

### B. Herramientas y compilación

El proyecto utiliza C++23 y raylib para la aplicación nativa. CMake configura el ejecutable y los programas de pruebas; el `Justfile` ofrece comandos cortos para las configuraciones nativa, web y de despliegue. En la versión web, Emscripten compila el mismo código a WebAssembly y precarga los assets; JavaScript conserva la persistencia del navegador y conecta la interfaz web con la API. El backend está escrito en módulos Node.js y se organiza para funciones Lambda, almacenamiento y ranking.

### C. Flujo de datos de una partida

La separación de módulos puede observarse también como una secuencia de datos. Un nivel se carga desde JSON y se convierte en un objeto de catálogo. La escena de construcción crea un modelo editable, pero no necesita conocer la implementación de la narrow phase. Al comenzar el intento, el modelo se copia y se enriquece con cuerpos, formas y joints. Durante la ejecución, el mundo físico produce posiciones y eventos; la escena de juego los utiliza para dibujar, comprobar la meta y acumular estadísticas. Finalmente, el módulo de puntuación consume únicamente esas estadísticas, lo que evita que la interfaz invente resultados físicos.

Tabla III: Responsabilidad de las principales capas de ejecución.

| Capa | Responsabilidad |
|---|---|
| Catálogo | Leer niveles, datos de usuario y configuraciones |
| Build | Editar celdas, piezas, conectores y orientación |
| Assembler | Crear cuerpos, formas, masas y joints |
| World | Integrar, detectar, resolver y producir eventos |
| Game rules | Meta, pasajeros, activaciones y estadísticas |
| Presentation | Cámara, texturas, audio, UI y partículas |
| Services | Score compartido, grabaciones, ranking y web |

El orden es relevante. El renderizador no debe construir una pieza física a partir de la textura, porque una textura no contiene masa, centro de giro ni conectores. De manera análoga, el sistema de puntuación no debe inferir la cantidad de piezas a partir de lo que permanece visible en pantalla: usa los identificadores y estadísticas que conserva la sesión. Esta disciplina de dependencias hace que sea posible probar una capa sin abrir una ventana.

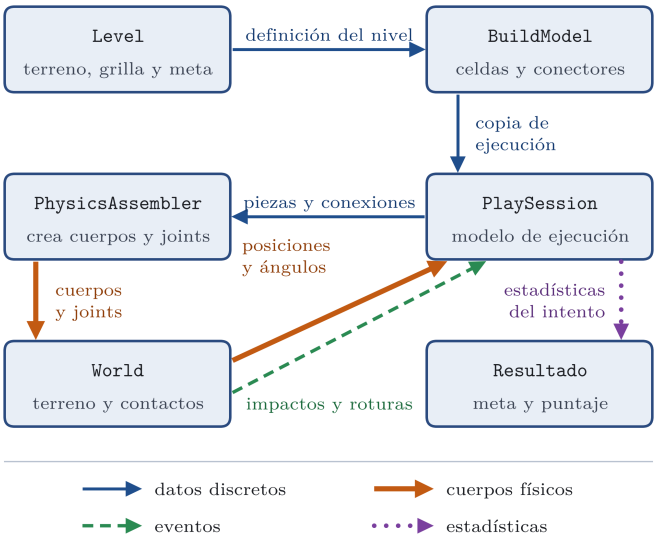



Figura 5: Flujo de datos desde el nivel hasta el resultado de la partida.

## IV. Geometría y representación gráfica

### A. Espacios de coordenadas

La construcción y la simulación utilizan representaciones distintas. Para una celda $\mathbf{c}$, un origen $\mathbf{o}$ y tamaño de celda $s$, la posición global puede expresarse como

$$\mathbf{p}_{world} = \mathbf{o} + s\mathbf{c}, \qquad \mathbf{c} = \left\lfloor \frac{\mathbf{p}_{world} - \mathbf{o}}{s} \right\rfloor. \tag{3}$$

La grilla necesita una dirección entera y reglas de ocupación; el mundo físico necesita posiciones reales, ángulos y velocidades. La cámara transforma del mundo a la pantalla y aplica seguimiento y límites del nivel. Mantener estas conversiones explícitas permite editar una pieza sin confundir su celda con el centro de su forma física.

Para un punto local $q$ de una pieza, su posición global (‘world position‘) se obtiene con una ‘local-to-world transform‘:

$$p_{world} = x + R(\theta)q, \qquad R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{4}$$

La transformación inversa usa $R(\theta)^T$. Esta distinción aparece en dos sitios: al dibujar una textura centrada en una pieza y al convertir un anclaje local de un joint a la posición actual del cuerpo. Un error en la dirección de la rotación puede dejar el sprite aparentemente correcto, pero aplicar el joint en un punto diferente al de la forma física.

### B. Terrenos y curvas Catmull–Rom

Los niveles pueden almacenar segmentos lineales o puntos de control para una curva. Para cuatro puntos $p_0, p_1, p_2, p_3$, el tramo Catmull–Rom usado por `Spline.cpp` se evalúa como

$$\begin{aligned} S(t) = \frac{1}{2}\big[&2p_1 + (-p_0 + p_2)t \\ &+ (2p_0 - 5p_1 + 4p_2 - p_3)t^2 \\ &+ (-p_0 + 3p_1 - 3p_2 + p_3)t^3\big]. \end{aligned} \tag{5}$$

El muestreo uniforme utiliza una resolución fija. El muestreo adaptativo comienza con el segmento entre dos puntos de control, evalúa puntos intermedios y lo subdivide si la desviación respecto de la cuerda supera una tolerancia. La implementación limita la profundidad de refinamiento a 12 niveles. De esta manera, los tramos casi rectos usan pocos vértices y las zonas de mayor curvatura reciben más muestras.

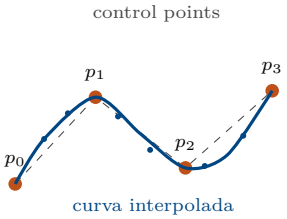



Figura 6: Control points y curva Catmull–Rom antes del muestreo para la malla.

### C. Validación y triangulación

Antes de construir una malla se comprueba que el contorno tenga al menos tres vértices, que ninguna arista sea menor que 0.5 unidades, que no existan intersecciones entre aristas no adyacentes y que el área absoluta sea al menos 1 unidad cuadrada. La prueba de intersección se basa en orientaciones mediante el producto cruzado y contempla también el caso de puntos sobre una arista.

**Algoritmo 2** Validación de un contorno de terreno.

**Require:** Polígono $P$ con $n$ vértices
**if** $n < 3$ **then**
  **return** Contorno inválido
**end if**
**for** cada arista de $P$ **do**
  **if** su longitud es menor que el umbral **then**
    **return** Arista corta
  **end if**
**end for**
**for** cada par de aristas no adyacentes **do**
  **if** las aristas se intersectan **then**
    **return** Autointersección
  **end if**
**end for**
**if** el área absoluta es menor que el umbral **then**
  **return** Área insuficiente
**end if**
**return** Contorno válido

Los polígonos válidos se triangulan por recorte de orejas. Se conserva una lista de índices, se busca un vértice convexo cuyo triángulo no contenga otros vértices, se elimina la oreja y se repite el procedimiento. La implementación verifica posteriormente que haya exactamente $n - 2$ triángulos, que todos tengan orientación positiva y que la suma de sus áreas coincida con el área del polígono. Esta última comprobación evita aceptar una triangulación incompleta.

**Algoritmo 3** Triangulación por recorte de orejas.

**Require:** Polígono válido $P$ con $n$ vértices
Orientar la lista de índices en sentido antihorario
$remaining \leftarrow [0, 1, \ldots, n-1]$
**while** $|remaining| > 3$ **do**
  $clipped \leftarrow false$
  **for** cada índice $i$ en $remaining$ **do**
    $prev, current, next \leftarrow$ vecinos de $i$
    **if** Cross$(prev, current, next) \leq \epsilon$ **then**
      Continuar con el siguiente vértice
    **end if**
    **if** otro vértice de $remaining$ está dentro de $(prev, current, next)$ **then**
      Continuar con el siguiente vértice
    **end if**
    Agregar triángulo $(prev, current, next)$
    Eliminar $current$ de $remaining$
    $clipped \leftarrow true$; **break**
  **end for**
  **if** no se recortó ninguna oreja **then**
    **return** Triangulación fallida
  **end if**
**end while**
Agregar el triángulo formado por los tres índices restantes
**if** cantidad de triángulos $\neq n-2$ o áreas no coinciden **then**
  **return** Triangulación fallida
**end if**
**return** Lista de índices triangulados

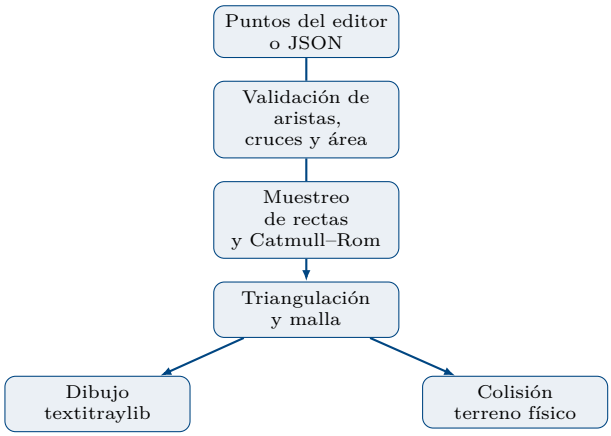



Figura 7: El terreno validado alimenta tanto al renderizado como a la física.

La Fig. 7 resume una decisión importante: el contorno no se dibuja por una ruta y se colisiona por otra independiente. Una misma secuencia muestreada y triangulada alimenta ambas etapas, reduciendo el riesgo de que el jugador vea una superficie que no coincide con el obstáculo físico.

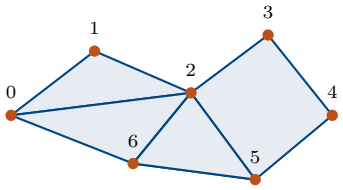



Figura 8: Ejemplo de la malla triangulada que se reutiliza para renderizado y colisión.

La validación de cruces compara cada par de aristas no adyacentes, por lo que su costo directo es $O(n^2)$. La triangulación por orejas también puede alcanzar un costo cuadrático, pues en cada eliminación busca si otros vértices están dentro del triángulo candidato. Para los niveles del proyecto esta decisión favorece la simplicidad y permite devolver diagnósticos visuales concretos: aristas marcadas, vértices problemáticos y una causa como "arista corta" o "autointersección". Un algoritmo más sofisticado podría mejorar el peor caso, pero no necesariamente ofrecería una mejor herramienta para el autor del nivel.

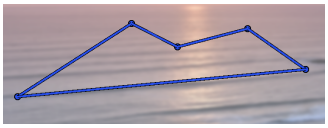
El muestreo adaptativo sigue una lógica recursiva. Se toma una cuerda entre los extremos de un subtramo, se evalúan puntos intermedios de la curva y se calcula la distancia entre la curva y la interpolación lineal. Si el máximo error supera la tolerancia, el intervalo se divide; de lo contrario, se conserva su extremo. El límite de profundidad evita que una entrada patológica produzca recursión indefinida.

**Algoritmo 4** Muestreo adaptativo de un tramo Catmull–Rom.

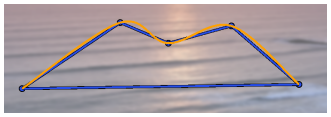
**Require:** Extremos $a, b$, puntos de control, tolerancia $\varepsilon$
Evaluar la curva en $t = 0.25$, $0.5$ y $0.75$
Calcular el error respecto de la cuerda $\overline{ab}$
**if** error máximo $> \varepsilon$ y no se alcanzó la profundidad límite **then**
  Dividir el intervalo en dos subtramos
  Aplicar este algoritmo a ambos subtramos
**else**
  Agregar $b$ a la secuencia de muestras
**end if**

### *D. Renderizado 2D*

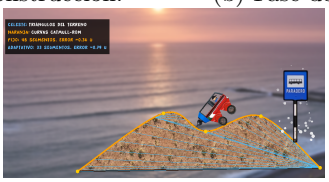
La cámara sigue la posición del vehículo y restringe su centro a los límites del nivel. Las piezas se dibujan aplicando la transformación de su celda, orientación y textura; los pasajeros y efectos se renderizan a partir de eventos de juego. El fondo utiliza capas con desplazamiento parallax, mientras que las partículas se disparan en impactos, explosiones, roturas y victoria. Raylib proporciona ventana, entrada, texturas, audio y rasterización; el proyecto calcula explícitamente las posiciones y los vértices de sus objetos.



(a) Fase de construcción.



(b) Fase de simulación.



(c) Pantalla de resultados.

Figura 9: Construcción geométrica del terreno y reutilización de la malla

## V. Motor de física

### A. *Rigid bodies y paso temporal*

Cada cuerpo dinámico conserva posición $x$, orientación $\theta$, velocidad lineal $v$, velocidad angular $\omega$, masa $m$, inversa de masa y momento de inercia $I$. Para una caja homogénea de ancho $w$ y alto $h$, se utiliza la aproximación

$$I = \frac{1}{12} m(w^2 + h^2). \tag{6}$$

El proyecto distingue los siguientes *rigid-body shapes*: `Box`, `Circle`, `Capsule`, `Triangle`, `Edge` y `TerrainMesh`. Los dos últimos representan geometría estática del terreno y no deben recibir integración dinámica. Para las formas dinámicas, las expresiones de referencia son:

$$I_{circle} = \frac{1}{2} m r^2, \tag{7}$$

$$I_{box} = \frac{1}{12} m(w^2 + h^2). \tag{8}$$

Para una *capsule*, una aproximación compositiva suma la contribución del rectángulo central y de los dos semicírculos, trasladando cada componente a su centro común mediante el teorema de ejes paralelos:

$$I_{capsule} = \sum_k \left( I_k + m_k d_k^2 \right). \tag{9}$$

Para un polígono uniforme, el área y el momento pueden obtenerse a partir de sus aristas. Definiendo $C_i = x_i y_{i+1} - x_{i+1} y_i$:

$$A = \frac{1}{2} \sum_i C_i, \tag{10}$$

$$I_z = \frac{1}{12} \sum_i C_i \big( x_i^2 + x_i x_{i+1} + x_{i+1}^2 \tag{11}$$

$$+ y_i^2 + y_i y_{i+1} + y_{i+1}^2 \big). \tag{12}$$

Estas fórmulas sirven como marco teórico para relacionar forma y rotación. La implementación concreta asigna los parámetros de cada pieza durante el ensamblaje y mantiene $invMass = 0$ e $invInertia = 0$ para cuerpos estáticos. Cuando una pieza agrega masa, el cuerpo actualiza su masa, inversa de masa e inercia proporcionalmente.

Tabla IV: Parámetros físicos asociados a cada shape.

| Shape | Uso y parámetro principal |
|---|---|
| `Circle` | Ruedas; radio $r$ y centro |
| `Box` | Cabinas y bloques; ancho $w$ y alto $h$ |
| `Capsule` | Piezas alargadas; radio y longitud |
| `Triangle` | Triángulos de soporte o terreno |
| `Edge` | Segmento estático para casos de borde; el terreno cargado usa triángulos |
| `TerrainMesh` | Conjunto estático de triángulos |

La configuración usada por `PlaySession` se resume en la Tabla V. Estos valores son parte del comportamiento reproducible del juego y no deben confundirse con las constantes teóricas de las ecuaciones anteriores.

Tabla V: Parámetros físicos implementados en la sesión de juego.

| Parámetro | Valor o comportamiento |
|---|---|
| Gravity | $(0, 392)$; equivale a $9.8 \times 40$ en unidades del juego |
| `subStepCount` | 10 subpasos por frame |
| `velocityIterations` | 8 iteraciones |
| `positionIterations` | 3 iteraciones |
| Linear damping | 0.05, aplicado exponencialmente |
| Angular damping | 0.65, aplicado exponencialmente |
| Friction | 1.25 por material, combinación geométrica |
| Restitution | 0.2 por material |
| First landing protection | 0.35 s después del primer contacto con terreno |
| Joint break threshold | 30 multiplicado por ‘break resistance‘ |

Durante un subpaso, el motor aplica fuerzas y gravedad, amortigua las velocidades y luego integra:

$$v_{t+\Delta t} = v_t + (F/m + g)\Delta t, \qquad x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t, \tag{13}$$

$$\theta_{t+\Delta t} = \theta_t + \omega_{t+\Delta t}\Delta t. \tag{14}$$

El mundo divide el frame en varios subpasos. En cada uno actualiza AABB, detecta colisiones y ejecuta iteraciones de restricciones. Esta elección reduce errores cuando una pieza se mueve rápido o cuando varios cuerpos están unidos.

El estado rotacional también interviene en la velocidad de un punto de contacto. En notación 2D, si $r$ es el brazo desde el centro de masa, la velocidad de ese punto puede expresarse como

$$v_p = v + \omega\,(-r_y, r_x). \tag{15}$$

Por eso un impacto lejos del centro puede producir poca traslación pero una rotación grande. Esta relación explica por qué el mismo impulso aplicado cerca de un anclaje o en el extremo de una pieza genera comportamientos distintos, y por qué el análisis estructural necesita considerar un brazo de palanca.

### B. Detección de colisiones

La *broad phase* genera pares potenciales usando una ‘bounding box‘, específicamente una ‘AABB‘ (‘axis-aligned bounding box‘). La *narrow phase* recibe cada par, descarta los que no se intersectan y construye un *manifold* para los restantes. El motor contiene rutas para las distintas combinaciones entre ‘Box‘, ‘Circle‘, ‘Capsule‘, ‘Triangle‘ y la geometría estática del terreno. Aunque el código conserva una representación ‘TerrainMesh‘, dicha malla está compuesta por triángulos; cada triángulo se incorpora como una unidad de colisión. Por ello, un contacto entre una pieza poligonal y el terreno termina procesándose como una pareja entre una forma convexa de la pieza y un triángulo del terreno, es decir, mediante la misma familia de pruebas poligonales que un caso *polygon–polygon*. Para terrenos abiertos, el código conserva además un ‘terrain chain‘ formado por segmentos, utilizado cuando el nivel no puede cerrarse como un polígono.

Para polígonos convexos, el ‘Separating Axis Theorem‘ (SAT, teorema del eje separador) busca un eje sobre el que las proyecciones no se solapen. Si existe un eje separador, no hay colisión; si todos los ejes se solapan, el menor solapamiento proporciona una normal y una profundidad de penetración.

Cuando la forma produce más de un punto de contacto, se recorta la cara incidente contra los planos de la cara de referencia siguiendo el algoritmo de Sutherland–Hodgman. El resultado es un conjunto pequeño de puntos que permite aplicar impulsos de manera estable.

La implementación distribuye la *narrow phase* en casos especializados, en lugar de convertir todas las formas a una única representación genérica. La Tabla VI relaciona algunos casos representativos con sus fuentes. En los niveles cerrados, la geometría del terreno se triangula y cada triángulo se utiliza como unidad estática; por eso los casos contra ‘TerrainMesh‘ terminan comparando la forma dinámica con triángulos.

Tabla VI: Ejemplos del dispatch de colisiones en la narrow phase.

| Caso | Fuente representativa |
|---|---|
| ‘Circle–Circle‘ | `CircleOnCircle.cpp` |
| ‘Box–Box‘ | `BoxOnBox.cpp` |
| ‘Box–Circle‘ | `BoxOnCircle.cpp` |
| ‘Capsule–Capsule‘ | `CapsuleOnCapsule.cpp` |
| ‘Polygon–Polygon‘ | `PolygonOnPolygon.cpp` |
| ‘Triangle–TerrainMesh‘ | `TriangleOnTerrainMesh.cpp` |
| ‘Circle–TerrainMesh‘ | `CircleOnTerrainMesh.cpp` |
| ‘Capsule–TerrainMesh‘ | `CapsuleOnTerrainMesh.cpp` |

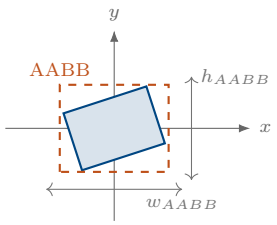



Figura 10: Ejemplo de una forma y su AABB utilizada en la *broad phase*.

La AABB es una aproximación económica: no representa exactamente la forma rotada, pero permite descartar pares que están separados sin ejecutar la narrow phase. Por ello, una falsa coincidencia de AABB no es todavía una colisión; solo significa que el par debe continuar hacia el detector específico de sus shapes.

**Algoritmo 5** Detección *circle–circle*.

**Require:** Centros $c_A, c_B$ y radios $r_A, r_B$
$\Delta \leftarrow c_B - c_A$
$d^2 \leftarrow \Delta \cdot \Delta$
$r \leftarrow r_A + r_B$
**if** $d^2 > r^2$ **then**
  **return** Sin colisión
**end if**
**if** $d^2 > \epsilon$ **then**
  $d \leftarrow \sqrt{d^2}$; $n \leftarrow \Delta / d$
**else**
  $d \leftarrow 0$; $n \leftarrow (1, 0)$
**end if**
$penetration \leftarrow r - d$
$p \leftarrow c_A + n(r_A - penetration/2)$
**return** $Manifold(n, penetration, p)$

El caso $d^2 \leq \epsilon$ evita normalizar un vector prácticamente nulo cuando los centros coinciden. La normal elegida es un fallback determinista; no pretende descubrir una dirección física única, que no existe en ese caso degenerado.

La tabla siguiente resume el papel de cada etapa del pipeline. No todas las formas requieren el mismo procedimiento geométrico, pero todas terminan en la interfaz común de un manifold.

Tabla VII: Etapas de detección de contactos.

| Etapa | Salida |
|---|---|
| Broad phase | Pares cuyos AABB se superponen |
| Narrow phase | Existencia de colisión, normal y profundidad |
| Clipping | Uno o más puntos de contacto |
| Manifold | Contactos listos para el solver |
| Postproceso | Impulsos e impactos asociados a piezas |

**Algoritmo 6** Narrow phase para dos polígonos convexos.

**Require:** Polígonos convexos $A$ y $B$
$minDepth \leftarrow +\infty$; $bestAxis \leftarrow \varnothing$
**for** cada arista de $A$ y de $B$ **do**
  $axis \leftarrow$ normal unitaria de la arista
  $I_A \leftarrow \textsc{Project}(A, axis)$
  $I_B \leftarrow \textsc{Project}(B, axis)$
  $min_I \leftarrow \min(I_A.max, I_B.max)$
  $max_I \leftarrow \max(I_A.max, I_B.max)$
  $overlap \leftarrow min_I - max_I$
  **if** $overlap \leq 0$ **then**
    **return** Sin colisión
  **end if**
  **if** $overlap < minDepth$ **then**
    $minDepth \leftarrow overlap$; $bestAxis \leftarrow axis$
  **end if**
**end for**
Orientar $bestAxis$ de $A$ hacia $B$
Elegir *reference edge* e *incident edge*
Recortar el *incident edge* con Sutherland–Hodgman
Conservar puntos detrás del *reference face*
**return** $Manifold(bestAxis, minDepth, points)$

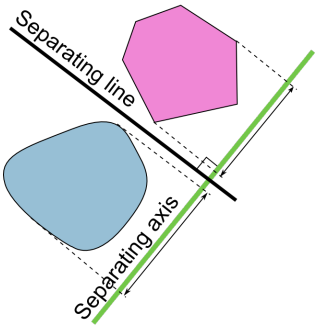



Figura 11: SAT: los polígonos se proyectan sobre ejes candidatos y se conserva el menor overlap.

**Algoritmo 7** Clipping de un segmento contra un plano.

**Require:** Segmento $SE$ y plano $(n, d)$
$insideS \leftarrow n \cdot S - d \leq 0$
$insideE \leftarrow n \cdot E - d \leq 0$
**if** $insideE$ y no $insideS$ **then**
  Agregar $\textsc{Intersection}(S, E, n, d)$
**else if** $insideE$ y $insideS$ **then**
  Agregar $E$
**else if** $insideS$ y no $insideE$ **then**
  Agregar $\textsc{Intersection}(S, E, n, d)$
**end if**
**return** Lista de vértices recortada

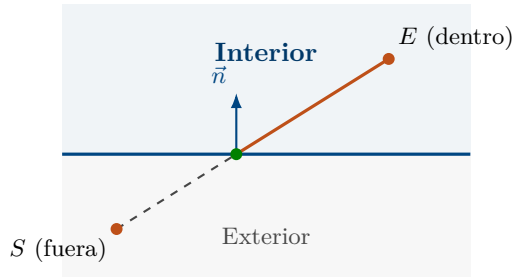



Figura 12: Clipping de un segmento $SE$ contra un plano: se evalúa la posición de $S$ y $E$, agregando el punto de intersección cuando el segmento cruza el plano.

### C. Respuesta por impulsos

Sea $n$ la normal del contacto y $r_A, r_B$ los brazos desde cada centro de masa hasta el punto. La velocidad relativa del punto de contacto incluye las contribuciones lineal y angular. Una forma compacta de expresar el impulso normal es

$$j_n = \frac{-(1+e)v_{rel} \cdot n}{m_A^{-1} + m_B^{-1} + K_{rot}}, \tag{16}$$

donde $e$ es la restitución y $K_{rot}$ reúne los términos de rotación producidos por los brazos de palanca. El impulso se aplica en sentidos opuestos a ambos cuerpos, modificando tanto su velocidad lineal como su velocidad angular.

La fricción se calcula con una dirección tangente al contacto. Su impulso se limita por el coeficiente de fricción y la magnitud del impulso normal, evitando que el solver produzca una fuerza tangencial físicamente desproporcionada. Las restricciones de contacto y de joints se resuelven iterativamente: primero se corrigen velocidades y luego se corrigen posiciones para reducir la penetración.

Para un cuerpo con posición angular, el denominador efectivo del impulso incluye el efecto de que el impulso pueda cambiar la rotación. Si $r_A \times n$ y $r_B \times n$ son los brazos escalares respecto de la normal, una forma habitual de escribirlo es

$$K_{rot} = I_A^{-1}(r_A \times n)^2 + I_B^{-1}(r_B \times n)^2. \tag{17}$$

Cuando el contacto se encuentra cerca del centro de masa, estos términos son pequeños; cuando se encuentra lejos, la misma magnitud de impulso produce una variación angular mayor. Esta relación también explica la sensibilidad de una estructura alta: el torque crece con la distancia entre el centro de masa y el contacto.

En la práctica, el solver no aplica una única fórmula a toda la escena y termina. Guarda impulsos acumulados dentro de las iteraciones, actualiza velocidades y vuelve a resolver los contactos que todavía tienen error. Este método, conocido como impulso secuencial, se relaciona con la arquitectura de motores como Box2D [2], [3].

### D. Pipeline de un paso

El algoritmo que organiza `World::Step` puede resumirse como sigue:

**Algoritmo 8** Paso de simulación física.

1: Procesar eventos pendientes y preparar el estado estructural
2: **for** cada subpaso **do**
3: Integrar los *rigid bodies* y actualizar sus AABB
4: Obtener pares candidatos mediante la *broad phase*
5: Detectar contactos en la *narrow phase*
6: Guardar cada contacto válido como un *manifold*
7: **for** cada iteración de velocidad **do**
8: Resolver *joints* y aplicar impulsos normales y tangenciales de cada *manifold*
9: **end for**
10: **for** cada iteración de posición **do**
11: Corregir *joints* y la penetración descrita por cada *manifold*
12: **end for**
13: Usar los impulsos acumulados en los *manifolds* para registrar impactos
14: **end for**
15: Aplicar impulsos pendientes, evaluar el stress de los *joints* y hacer cleanup

El 'manifold' es la estructura intermedia que conecta la detección con la respuesta: se crea en la 'narrow phase', se consulta durante las iteraciones del 'velocity solver', se reutiliza para la 'position correction' y finalmente permite asociar los impulsos con un 'impact event'. El motor omite la detección de colisión entre cuerpos conectados por un joint cuando la caché de conectividad indica que no debe resolverse como contacto. Así, una conexión estructural no se convierte inmediatamente en una colisión interna que contradiga su restricción.

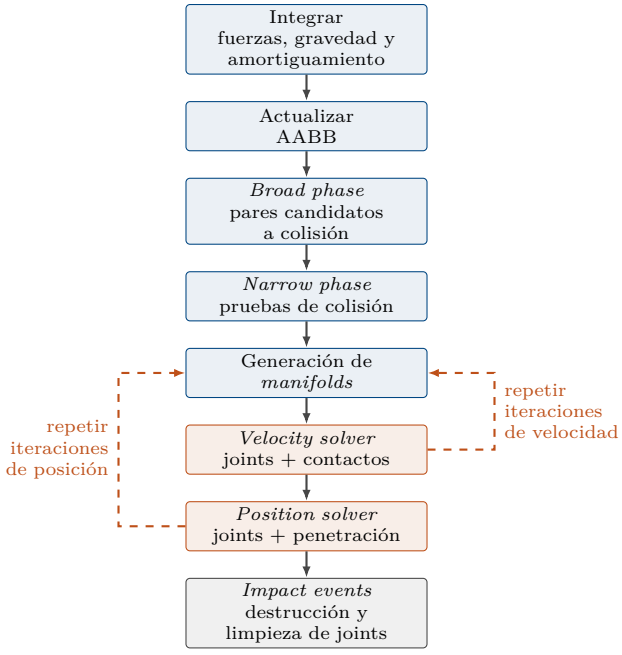



Figura 13: Pipeline de simulación de un frame del motor físico. Los *manifolds* generados en la *narrow phase* se reutilizan durante las iteraciones de velocidad y posición.

### *E. Joints y análisis estructural*

Una rueda utiliza un *revolute joint* (joint de revolución): sus anclajes deben coincidir, pero se permite la rotación relativa. Un *weld joint* (joint soldado) restringe tanto la posición relativa como el ángulo. En ambos casos, el solver genera impulsos que corrigen el error de la restricción.

En un 'revolute joint', la restricción principal es la coincidencia entre los dos anchors transformados a 'world space'. Si $a_A$ y $a_B$ son esos anchors, el error de posición es

$$C_{pos} = a_B - a_A. \tag{18}$$

El solver intenta reducir sus dos componentes, pero no bloquea la diferencia angular entre los cuerpos. En un 'weld joint' se añade un error angular:

$$C_{angle} = \text{wrap}(\theta_B - \theta_A - \theta_{relative}). \tag{19}$$

Por ello, el 'weld joint' necesita una corrección angular adicional, mientras que el 'revolute joint' conserva un grado de libertad rotacional.

La rotura no depende solamente de un umbral global aplicado a todos los joints. En cada paso se reconstruye un 'joint graph' cuyos nodos son 'rigid bodies' y cuyas aristas son conexiones. Para evaluar un joint, se lo excluye temporalmente y se exploran las 'connected components' de cada extremo. Si los extremos quedan en componentes distintas, el joint actúa como un 'bridge joint' y el 'local load' incluye la masa del 'dependent component' que cuelga de él. También se consideran el 'center of mass', el 'lever arm' y factores de 'damping' asociados a masa y torque. Este cálculo busca que un joint local soporte principalmente la estructura que realmente sostiene.

Una unión se marca para rotura cuando el esfuerzo local supera su resistencia. La limpieza elimina el joint del mundo, invalida la caché de conectividad y genera un evento visual en la posición intermedia de los cuerpos. Las explosiones constituyen un caso adicional: aplican impulsos con atenuación lineal según la distancia y pueden activar TNT o romper conexiones afectadas.

Este procedimiento puede ilustrarse con una estructura de tres cuerpos $A$–$B$–$C$. Si se evalúa la unión entre $A$ y $B$, el algoritmo la excluye temporalmente y calcula las componentes restantes. Si $C$ queda conectado a $B$ pero no a $A$, la unión es un puente y puede cargar el componente que contiene a $C$. Si existe otra ruta entre ambos extremos, el joint no es un puente; su resistencia se amortigua para evitar que una conexión redundante soporte toda la masa como si fuera la única. Esta decisión produce roturas más locales que un umbral basado solamente en la masa total del vehículo.

**Algoritmo 9** Evaluación de carga local de un *joint*.

**Require:** *joint* $J$ y grafo de cuerpos y conexiones
Excluir temporalmente $J$ del grafo
Obtener la componente conectada a cada extremo de $J$
**if** las componentes son distintas **then**
    Marcar $J$ como puente
    Seleccionar el componente dependiente según gravedad y masa
**else**
    Marcar $J$ como conexión redundante
**end if**
Calcular centro de masa y brazo de palanca
Aplicar factores de *damping* de masa, torque y redundancia
**return** Carga local y resistencia ajustada

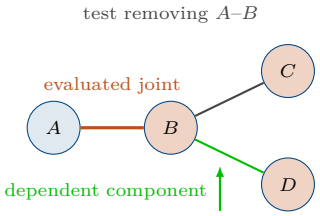



Figura 14: Ejemplo de un *joint* puente y la componente dependiente de A-B.

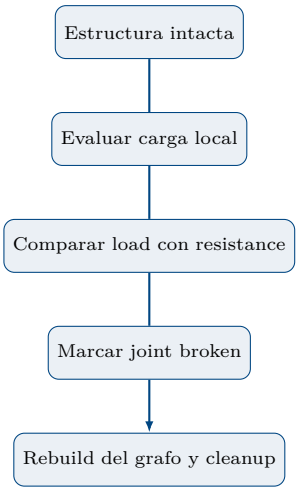



Figura 15: Flujo de decisión para la rotura estructural.

### F. Estabilidad y limitaciones del solver

El solver no busca invertir una matriz global en cada frame. Resuelve restricciones locales repetidamente, aproximando una solución global después de varias iteraciones. El número de iteraciones de velocidad y posición constituye un compromiso entre costo y estabilidad: pocas iteraciones son rápidas pero dejan más error, mientras que demasiadas pueden aumentar el tiempo de cada frame sin eliminar todos los problemas numéricos.

El código utiliza ‘exponential damping‘ dependiente del subpaso, en vez de restar una cantidad fija por frame. Así el efecto se mantiene más estable cuando cambia la cantidad de subpasos. También existe una protección breve en el primer contacto con el terreno para evitar que una construcción recién soltada falle por un impacto inicial no representativo. Estas decisiones son de gameplay y estabilidad, no afirmaciones de que el modelo sea una simulación física exacta.

La corrección posicional tiene un segundo objetivo: impedir que el error geométrico se acumule cuando dos cuerpos penetran ligeramente. La estabilización tipo Baumgarte introduce una velocidad correctiva proporcional a la penetración, pero se limita para no convertir una pequeña imprecisión en una explosión numérica. La construcción del manifold y el clipping siguen la formulación clásica de contacto poligonal [6], [5]; el proyecto la adapta a sus formas y a sus límites de precisión.

## VI. Sistemas de juego sobre la física

### A. Construcción y ensamblaje

El jugador no manipula directamente *rigid bodies* durante la construcción. `EditorGrid` controla límites y ocupación; `BuildSession` mantiene selección, orientación, colocación, extracción y reposicionamiento; `BuildModel` conserva las piezas y conexiones. Las definiciones de piezas determinan qué conectores son compatibles y cómo debe orientarse una pieza acoplada.

Al iniciar un intento, `PlaySession` crea el modelo de ejecución. `PhysicsAssembler` traduce cada pieza a un cuerpo, asigna su forma y masa, y crea joints para los conectores válidos. El terreno se incorpora como cuerpos estáticos a partir de la geometría ya validada. Esta conversión es el puente entre el diseño discreto que el jugador edita y el estado continuo que el solver integra.

### B. Activaciones y eventos

Las activaciones se mantienen separadas de la integración física mediante un registro. Un ventilador o una vela puede aportar propulsión; una explosión puede aplicar impulsos radiales, romper joints y activar otra pieza TNT; un impacto produce daño visual y puede generar una rotura. El renderizador consume estos eventos, pero no altera por sí mismo la posición de los cuerpos. Esta separación también permite que la reproducción de una partida lea estados y eventos con una cadencia definida.

### C. Editor de niveles, ranking y fantasma

El editor personalizado permite definir terreno, zona de construcción, meta, pasajeros y música. Los niveles se serializan en JSON y pueden validarse antes de publicarse. El cliente conserva datos locales de niveles y récords; la API ofrece funciones para identidad, sesiones, niveles, intentos, grabaciones y leaderboard. El scoring compartido reduce diferencias entre la puntuación local y la validada por el servidor.

Las partidas campeonas se capturan a 20 Hz, con límites de duración y tamaño indicados en el README del proyecto. El fantasma se reproduce leyendo la trayectoria registrada, no ejecutando nuevamente todas las decisiones del jugador. La versión WebAssembly conserva la lógica del juego y utiliza JavaScript para persistencia del navegador y comunicación con el sitio. La arquitectura completa de la versión web, con los componentes del navegador y los servicios desplegados en AWS, se muestra en la Fig. 16 del Apéndice E.

El backend separa operaciones de lectura y escritura para evitar que una consulta de ranking modifique el estado de un nivel. Entre las funciones disponibles se encuentran la creación y consulta de sesiones, listado y publicación de niveles, envío de intentos, consulta de grabaciones, renombrado de identidad y limpieza de grabaciones huérfanas. La infraestructura declara estos recursos en una plantilla y el script de despliegue ofrece tareas separadas para API y sitio. En un entorno local, sin embargo, el juego puede ejecutarse sin desplegar estos servicios, usando los datos de usuario y niveles incluidos.

Tabla VIII: Grupos de operaciones del backend.

| Grupo | Operaciones representativas |
|---|---|
| Health | Comprobar disponibilidad |
| Identity | Crear sesión y renombrar identidad |
| Levels | Listar, consultar, copiar y retirar niveles |
| Publication | Publicar un nivel validado |
| Attempts | Enviar y consultar intentos |
| Recordings | Subir, consultar campeón y limpiar huérfanos |
| Leaderboard | Consultar ranking por nivel |

Las rutas del backend comienzan con ‘/api/v1‘ y separan el ‘levelId‘ de la información del cuerpo. Algunos contratos relevantes son:

Tabla IX: Contratos resumidos de la API.

| **Ruta** | **Resultado principal** |
|---|---|
| `GET /health` | Estado 200 del servicio |
| `POST /session` | Crea identidad y cookie |
| `GET /levels/{levelId}` | Definición del nivel |
| `PUT /levels/{levelId}/attempts/{attemptId}` | Valida y almacena un intento |
| `GET /levels/{levelId}/leaderboard` | Ranking del nivel |
| `PUT .../recording` | Sube una grabación |
| `POST /levels` | Publica un nivel |

Las operaciones de escritura requieren el ‘Origin‘ configurado y la cabecera ‘x-combi-request: 1‘. Los errores se devuelven como un objeto con campo ‘error‘; las grabaciones se identifican mediante hash y se almacenan en S3, mientras que el estado de dominio se conserva en DynamoDB.

## VII. Evaluación

### A. Validación geométrica

El repositorio contiene el ejecutable de prueba de `proofs/terrain_geometry_proof.cpp` y el archivo `validacion_geometrica.csv`. La batería incluye cuadrados, polígonos cóncavos, cruces, contactos, aristas cortas, áreas pequeñas y contornos con pocos puntos. Los casos válidos producen $n-2$ triángulos: por ejemplo, el cuadrado produce dos, el polígono cóncavo produce tres y los niveles oficiales inspeccionados cumplen esta relación. Los casos inválidos son rechazados antes de la triangulación.

Tabla X: Casos representativos de la prueba geométrica.

| Caso | Vértices | Diagnóstico | Triángulos | Esperados |
|---|---|---|---|---|
| Cuadrado | 4 | válido | 2 | 2 |
| Cóncavo | 5 | válido | 3 | 3 |
| Cruce | 4 | autointersección | 0 | 2 |
| Toque | 5 | autointersección | 0 | 3 |
| Repetido | 4 | arista corta | 0 | 2 |
| Colineal | 3 | área pequeña | 0 | 1 |

La prueba de muestreo compara dos configuraciones de terreno: muestreo fijo y muestreo adaptativo, tanto en reposo como durante un recorrido. En las 27 repeticiones registradas en `benchmark_muestreo_terreno.csv`, el adaptativo reduce el tiempo de reposo y de recorrido de manera consistente en el escenario medido. La comparación debe interpretarse como una medición de ese nivel y hardware, no como una garantía universal: el número de puntos, la curvatura y la cantidad de contactos cambian el costo.

En particular, la mediana registrada es de 103.771 ms para el reposo con muestreo fijo y 60.824 ms con muestreo adaptativo, una reducción aproximada de 41.4%. En el recorrido, las medianas son 45.604 ms y 19.010 ms, respectivamente, una reducción aproximada de 58.3%. La existencia de una reducción en ambos casos es coherente con la intención del algoritmo: representar con menos segmentos las zonas donde la curva no necesita refinamiento. Estas cifras son evidencia del archivo de pruebas incluido, no una comparación entre computadores distintos.

### B. Complejidad y reproducibilidad de las mediciones

La validación de cruces y la triangulación por orejas tienen costo cuadrático en el número de vértices. La broad phase reduce el número de pares que llegan a la narrow phase, aunque su costo depende de la estrategia y de la cantidad de cuerpos. La resolución de contactos es aproximadamente proporcional al número de manifolds por el número de iteraciones de velocidad y posición. El análisis estructural explora componentes del grafo para cada joint y por ello también crece con cuerpos y conexiones.

Para reproducir una medición no basta con copiar su porcentaje. Deben conservarse el nivel de entrada, la cantidad de repeticiones, el modo de compilación, la configuración del solver y el archivo de salida. Los CSV dentro de `proofs/` registran los datos observados; el informe presenta las

medianas para compactar la discusión, mientras que el repositorio conserva cada repetición.

### C. Pruebas funcionales y físicas

La evaluación funcional se organiza alrededor del flujo de usuario. La construcción se prueba con piezas compatibles e incompatibles; la transición verifica que el número de cuerpos y joints corresponda al modelo; la simulación observa contactos, fricción, activaciones e impactos; la meta comprueba la entrada entre frames y la ventana de tres segundos; y el scoring se contrasta con vectores compartidos. Las pruebas físicas aisladas son preferibles a inferir el funcionamiento del motor únicamente a partir de una partida completa, porque permiten separar el costo del contacto, la propulsión y la topología estructural.

Los casos físicos mínimos que deberían repetirse después de modificar el motor son los siguientes:

Tabla XI: Casos físicos mínimos para regresión.

| Caso | Comportamiento que debe observarse |
|---|---|
| ‘Circle‘ sobre ‘Triangle‘ | Contacto, normal y separación estable |
| ‘Box‘ sobre terreno inclinado | Rotación, fricción y ausencia de penetración persistente |
| ‘Circle–Circle‘ | Manifold con normal y profundidad válidas |
| ‘revolute joint‘ | Anchors coincidentes y rotación relativa libre |
| ‘weld joint‘ | Posición y diferencia angular restringidas |
| ‘bridge joint‘ | Carga dependiente y rotura local |
| Explosión | Atenuación por distancia, impulso y cleanup |

Cada caso debe registrar el escenario, la configuración del ‘World‘, el número de subpasos e iteraciones, el estado inicial y una observación cuantitativa. Esto evita que una prueba visual de una partida sustituya a una prueba aislada del contacto o de la restricción.

Tabla XII: Matriz compacta de pruebas reproducibles.

| Prueba | Entrada | Observación |
|---|---|---|
| Geometría | Polígonos válidos e inválidos | Diagnóstico, área y triángulos |
| Muestreo | Curvas del nivel de referencia | Tiempo y cantidad de muestras |
| Contacto | Cuerpo sobre terreno | Penetración, estabilidad y fricción |
| Construcción | Piezas y conectores | Aceptación y ensamblaje |
| Meta | Conductor cruzando el área | Inicio y final de ventana |
| Rotura | Joint con carga o explosión | Eliminación y evento |
| Scoring | Estadísticas del intento | Puntaje local y compartido |

## VIII. Reproducción del proyecto

El proyecto requiere un compilador C++23, CMake, raylib y, opcionalmente, `just`. En Linux, el README propone instalar los paquetes del compilador, CMake, raylib y just; en macOS, raylib y CMake pueden instalarse con Homebrew. Desde la raíz de `project/`, la reproducción nativa mínima es:

Tabla XIII: Controles principales observados en la sesión de juego.

| Entrada | Acción |
|---|---|
| `W,A,S,D` | Mover la cámara en modos de visualización |
| `R` / rueda | Rotar una pieza o cambiar su orientación |
| Click izquierdo | Seleccionar, colocar o arrastrar |
| Click derecho | Retirar una pieza o arrastrar la cámara, según el modo |
| `Enter` | Cambiar de modo o iniciar la acción contextual |
| `Escape` | Limpiar selección o cancelar |
| Teclas numéricas | Activar acciones visibles del ‘ActivationRegistry‘ |
| Flechas | Navegar opciones o selección contextual |

```
just build
just run
```

El binario queda en `build/native/ray`. CMake usa el estándar C++23, recoge los fuentes de `src/` y enlaza raylib con las bibliotecas gráficas del sistema. Los programas de prueba geométrica se construyen como objetivos adicionales en configuraciones nativas.

Las pruebas numéricas se construyen explícitamente como objetivos adicionales:

```
cmake --preset native-debug
cmake --build --preset native-debug \\
  --target terrain_geometry_proof terrain_sampling_proof
./build/native/terrain_geometry_proof
./build/native/terrain_sampling_proof
```

La primera prueba genera o comprueba los diagnósticos de validación y triangulación. La segunda utiliza `assets/levels/default.json`, compara muestreo fijo y adaptativo, crea un ‘World‘ con gravity $(0, 392)$, 10 subpasos, 8 ‘velocity iterations‘ y 3 ‘position iterations‘, y registra los resultados en los CSV de `proofs/`. El código utiliza 180 pasos de calentamiento, 240 pasos medidos y 9 repeticiones por configuración.

Para WebAssembly se instala y activa Emscripten, se compila con:

```
source /ruta/a/emsdk/emsdk_env.sh
just web
just serve
```

El servidor local expone `ray.html` en el puerto 6931. Es importante usar un servidor porque el navegador debe cargar el filesystem virtual y los assets; abrir el HTML directamente con `file://` no reproduce el entorno esperado. Las tareas `web-release`, `aws-deploy-api` y `aws-deploy-web` cubren respectivamente una compilación optimizada y el despliegue separado de backend y sitio. El despliegue AWS es una extensión del producto, no un requisito para ejecutar la versión local.

La estructura mínima que un lector debe inspeccionar es:

```
project/
  src/       # juego, geometria, fisica y renderizado
  proofs/    # pruebas numericas
  assets/    # texturas, audio, fuentes y niveles
  api/       # funciones del backend
  infra/     # infraestructura de despliegue
  CMakeLists.txt  Justfile  README.md
```

Para extender el proyecto, una nueva pieza debe añadir su definición, textura, forma física, conectores y comportamiento de activación; un nivel nuevo debe respetar el esquema JSON y pasar la validación del terreno. Después de cada modificación conviene repetir las pruebas geométricas, construir el ejecutable nativo y verificar una partida completa.

## IX. Discusión y conclusiones

El producto demuestra que un ciclo de juego recreativo puede servir como problema integrador de Computación Gráfica. La construcción sobre una grilla es fácil de explicar al usuario, pero exige una conversión precisa hacia cuerpos continuos. De forma análoga, una curva de terreno es una herramienta de autoría, una malla visible y una colección de obstáculos físicos al mismo tiempo. Si estas representaciones divergen, el jugador observa comportamientos que no puede razonar.

La decisión de implementar un motor propio permite exponer el proceso completo: broad phase, narrow phase, manifolds, impulsos, fricción, joints, corrección posicional y análisis estructural. El análisis del grafo no reemplaza a un solver estructural general, pero proporciona una regla local y comprensible para que una conexión soporte la estructura dependiente de ella. El sistema también muestra el costo de integrar física con reglas de juego: la victoria, el puntaje, las activaciones, las grabaciones y el ranking deben leer un estado consistente.

Como limitaciones, el solver es iterativo y está diseñado para la escala del juego, no para garantizar exactitud física en todos los regímenes. El costo de la broad phase, la cantidad de iteraciones, la discretización del terreno y los umbrales de rotura afectan el resultado. La evaluación geométrica está automatizada en mayor medida que la evaluación visual y de gameplay; una ampliación futura debería incorporar pruebas de regresión para escenas completas, instrumentación sistemática del motor y más casos de contacto.

En conclusión, *Tunea Tu Combi* cumple el objetivo de convertir una idea de construcción y reintento en un sistema reproducible que combina representación gráfica, geometría computacional, simulación de *rigid bodies* y servicios de juego. El documento y el repositorio permiten seguir el camino completo desde el nivel y la grilla hasta el cuerpo físico, el resultado de la partida y la compilación nativa o web.

## Appendix A
## Parámetros y archivos de referencia

La siguiente tabla indica dónde verificar las decisiones descritas en este informe.

Tabla XIV: Trazabilidad entre componentes y archivos del proyecto.

| Componente | Archivos principales |
|---|---|
| Construcción | `src/app/game/BuildModel.*`, `BuildSession.*`, `EditorGrid.*` |
| Terreno | `src/math/Terrain.*`, `Spline.*` |
| Ensamblaje | `src/physics/PhysicsAssembler.*`, `Part.*` |
| Paso físico | `src/physics/engine/step-pipeline/` |
| Estructura | `StructuralAnalysis.*`, `Joint.hpp` |
| Meta y score | `GoalWindowTracker.*`, `scoring/ScoreRules.*` |
| Web y API | `src/web/`, `api/`, `infra/` |
| Pruebas | `proofs/`, `shared/scoring-v1-vectors.json` |

## Appendix B
## Pseudocódigo de inspección

Para inspeccionar una modificación del motor se recomienda seguir esta secuencia, que coincide con la organización de los fuentes:

**Algoritmo 10** Inspección de una modificación del motor.

1: Construir un caso mínimo de cuerpos y *joints*
2: Ejecutar el paso con gravedad y subpasos conocidos
3: Revisar AABB y pares de *broad phase*
4: Revisar *manifold*, normal y profundidad
5: Revisar impulsos y corrección de posiciones
6: Revisar esfuerzo local y limpieza de *joints*
7: Comparar el resultado visual con el estado físico

Esta secuencia evita diagnosticar una falla de renderizado como si fuera una falla del solver, o una discrepancia de la meta como si fuera una colisión.

## Appendix C
## Esquema mínimo de un nivel

Un nivel combina geometría, zona de construcción, objetivo y contenido. La representación concreta puede evolucionar, pero conceptualmente contiene los siguientes grupos:

```
{
  "terrain": [
    {"type":"Line", "points":[...]},
    {"type":"CatmullRom", "points":[...]}
  ],
  "grid": {"origin":[x,y], "width":..., "height":...},
  "goal": {"position":[x,y], "size":[w,h]},
  "passengers": [{"hostId":..., "position":[x,y]}],
  "camera": {...},
  "music": "..."
}
```

Al cargarlo, el terreno pasa por validación y triangulación antes de incorporarse al mundo. La grilla restringe la construcción, pero no sustituye al terreno físico. La meta y los pasajeros se mantienen como reglas de juego que leen posiciones de los cuerpos durante la simulación.

## Appendix D
## Checklist de reproducción

La siguiente lista condensa el procedimiento que debe seguir un alumno que reciba el repositorio:

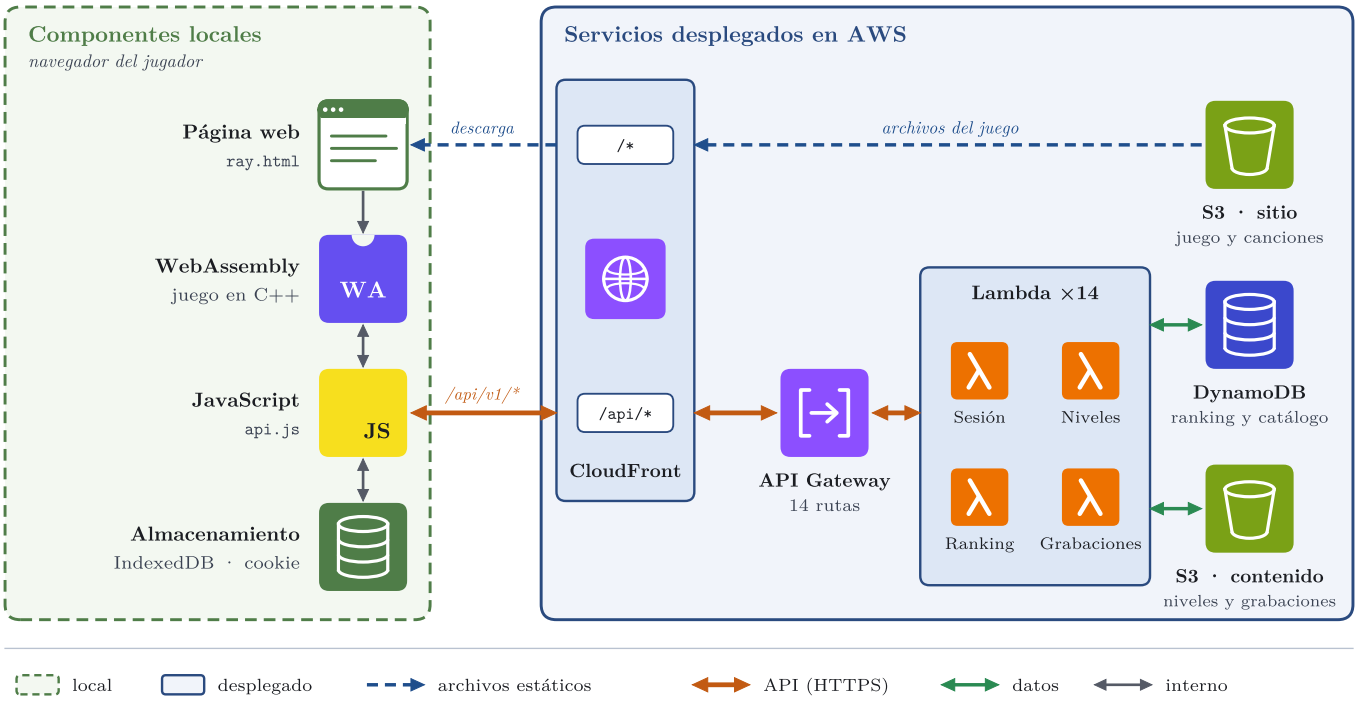



Figura 16: Arquitectura de la versión web y de los servicios auxiliares.

1) Instalar un compilador C++23, CMake, raylib y `just`.
2) Entrar al directorio `project/` y ejecutar `just build`.
3) Ejecutar `just run` y comprobar el menú, la selección de nivel y la construcción.
4) Construir una combi con conductor, estructura y al menos una forma de movimiento.
5) Iniciar una partida y comprobar que las piezas se convierten en cuerpos y joints.
6) Ejecutar las pruebas geométricas y revisar los CSV de `proofs/`.
7) Si se requiere la versión web, activar Emscripten y ejecutar `just serve`.
8) Para una extensión, modificar una sola capa, repetir las pruebas y observar una partida completa.

La instalación de servicios AWS no es necesaria para repetir la simulación local. Si se desea reproducir el despliegue, se deben configurar además las credenciales y recursos indicados en `infra/README.md`; por su naturaleza, esa parte depende de una cuenta externa y debe considerarse una ruta de entrega, no una dependencia del motor.

## Appendix E
## Arquitectura de la versión web

La Fig. 16 muestra los componentes locales del navegador y los servicios desplegados en AWS que sostienen la versión web.

## Referencias


[1] Equipo Tunea Tu Combi, "Repositorio del proyecto de Computación Gráfica," código fuente, 2026.
[2] E. Catto, "Iterative dynamics with temporal coherence," notas de GDC y documentación de Box2D, 2005–2024.
[3] E. Catto, "Box2D Lite: A fast and simple 2D physics engine," código y notas técnicas, 2006.
[4] D. Baraff, "An introduction to rigid body simulation," notas de curso, Carnegie Mellon University / Pixar, 2001.
[5] D. H. Eberly, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, 2nd ed. Morgan Kaufmann, 2010.
[6] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Communications of the ACM*, vol. 17, no. 1, pp. 32–42, 1974.
[7] E. Catmull and R. Rom, "A class of local interpolating splines," in *Computer Aided Geometric Design*, 1974, pp. 317–326.
[8] R. C. M. Ferreira, "raylib: A simple and easy-to-use library to enjoy videogame programming," documentación del proyecto, 2013–2026.
[9] Emscripten Contributors, "Emscripten compiler toolchain," documentación técnica, 2026.