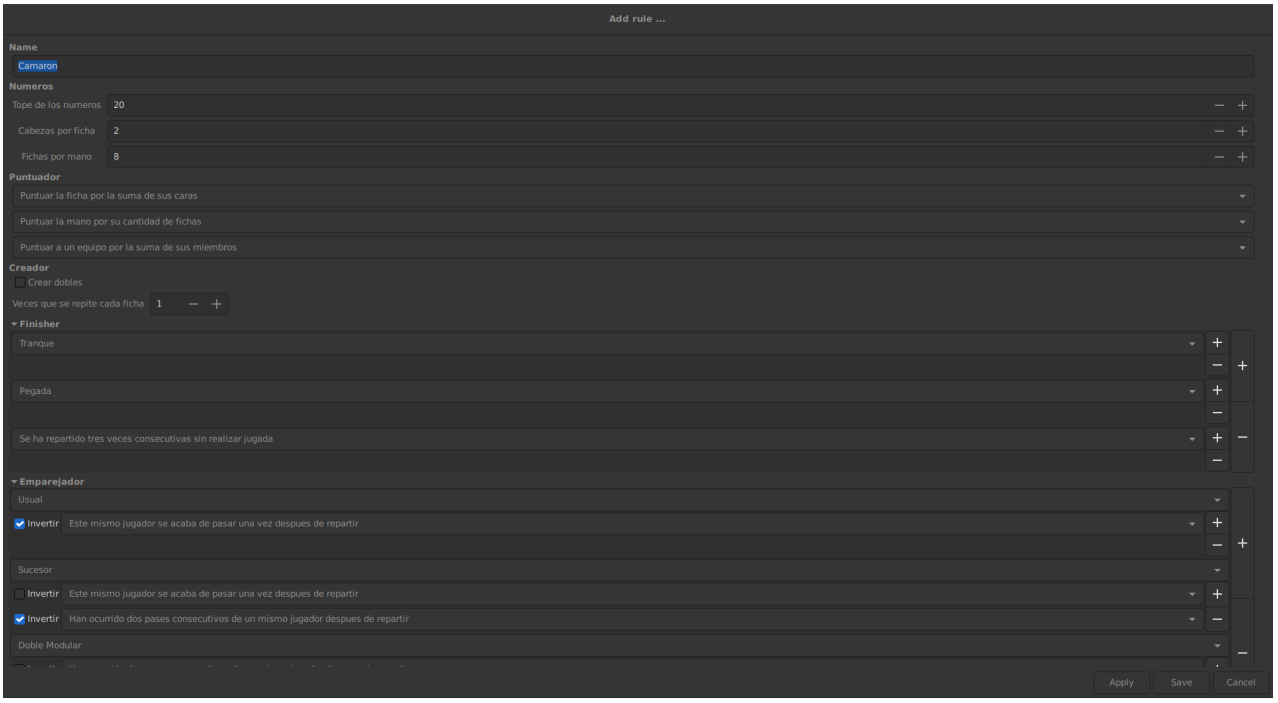
Descripción del proyecto (Frontend)

Sobre los datos del juego

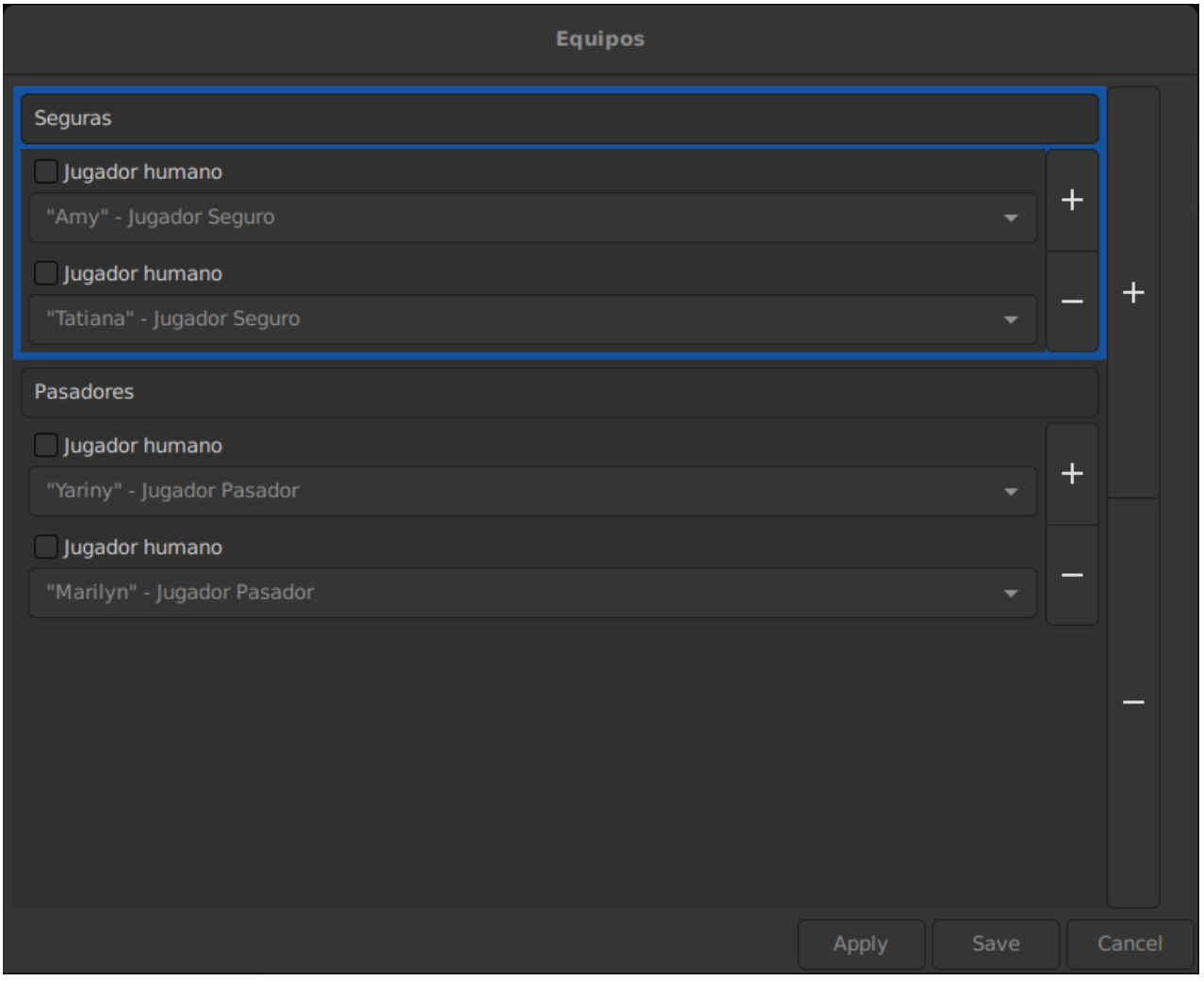
Los modelos y texturas del juego son archivos que no se modifican una vez que se incorporan al proyecto que asumí no necesitarían del control de versiones (aka. git), además de ocupar mucho espacio, pero son necesarios para la ejecución del proyecto. A tal efecto los subiré comprimidos como parte de una "release". Opcionalmente se puede bajar aquí. Descomprima y póngalo en 'data/models'

Estructura interna

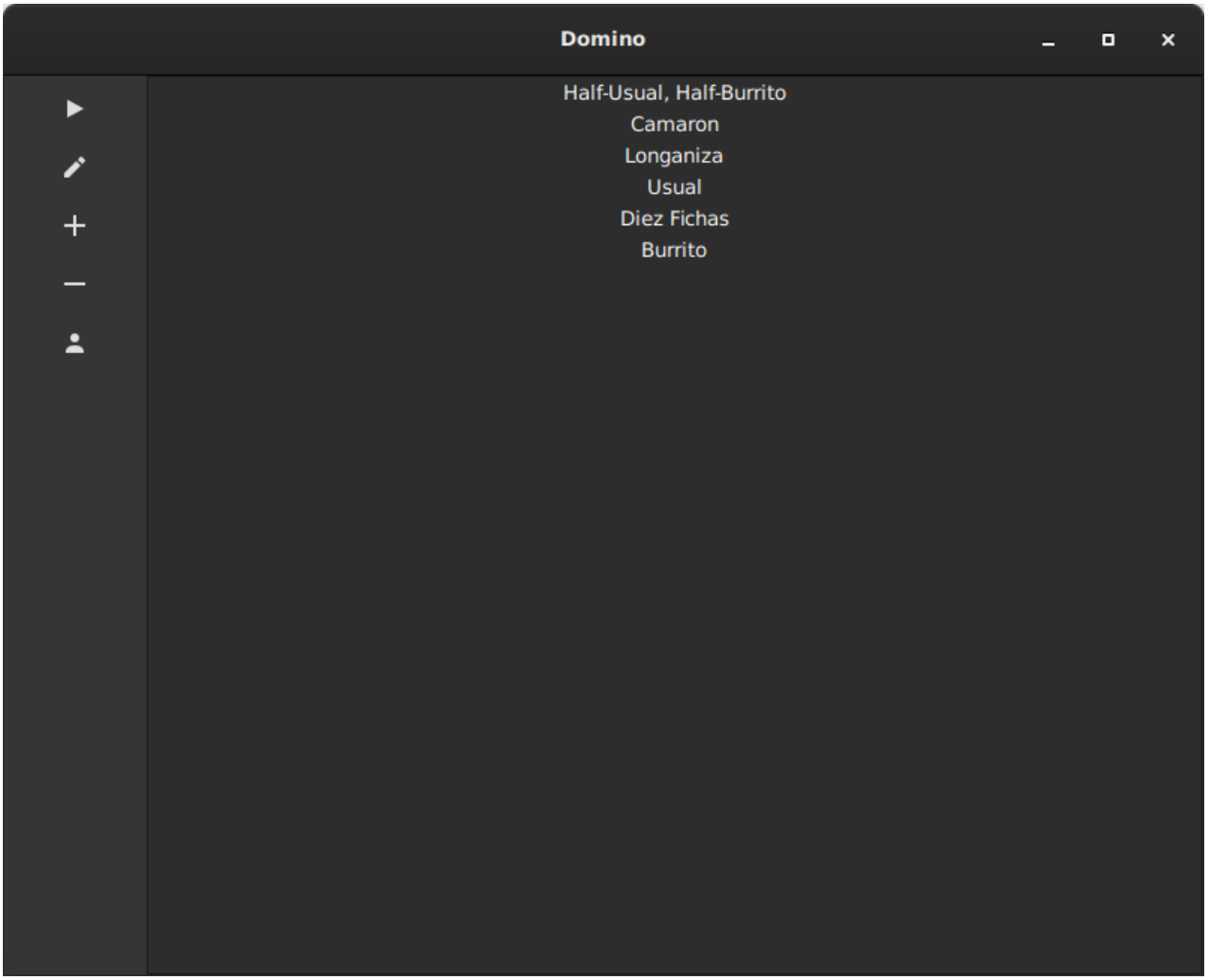
- `Frontend` : Aquí están todo lo que tiene el **frontend** (la parte gráfica) del proyecto
  - `Application` : punto de entrada a la aplicación. Implementa `Patch.Application` que a su vez implementa `Gtk.Application` que es necesaria para trabajar con **Gtk**.
  - `IListBoxRow` : representa un objecto que puede adicionarse a un objeto `ListBox`
    - `Rows`
      - `Rows.emparejador` : clase asociada al archivo "Emparejador.txt" de las reglas del juego
      - `Rows.finisher` : clase asociada al archivo "Finisher.txt" de las reglas del juego
      - `Rows.moverturno` : clase asociada al archivo "MoverTurno.txt" de las reglas del juego
      - `Rows.refrescador` : clase asociada al archivo "Refrescador.txt" de las reglas del juego
      - `Rows.repartidor` : clase asociada al archivo "Repartidor.txt" de las reglas del juego
      - `Rows.validador` : clase asociada al archivo "Validador.txt" de las reglas del juego
  - `ListBox` : widget que se usa para representar variadas opciones de configuración a lo largo del proyecto (de la parte gráfica)
  - `Message` : clase auxiliar usada para lanzar diálogos estilo "message box" (cuadro de diálogo)
  - `RuleDialog` : clase que encapsula el controlador de un regla. Desde este pueden editarse de forma más fácil las reglas del dominó



- `TeamDialog` : clase que encapsula el controlador de los equipos. Desde este pueden editarse de forma más fácil la disposición de los equipos y los diferentes jugadores



- `Window` : ventana principal desde la cual se accede al juego. Desde esta se puede acceder a las opciones de configuración de los equipos, las reglas y se pueden iniciar las partidas



- `Frontend.Patch`
  - class Application: Clase auxiliar para sortear un bug en la biblioteca **GioSharp** que causa un error interno cuando se lanza un evento de tipo **GLib.Application.Opened** (el marshaler recibe un puntero a un array de punteros a objetos nativos de type GObject y lo interpreta como un puntero a un objeto, mi clase sobrescribe es comportamiento)

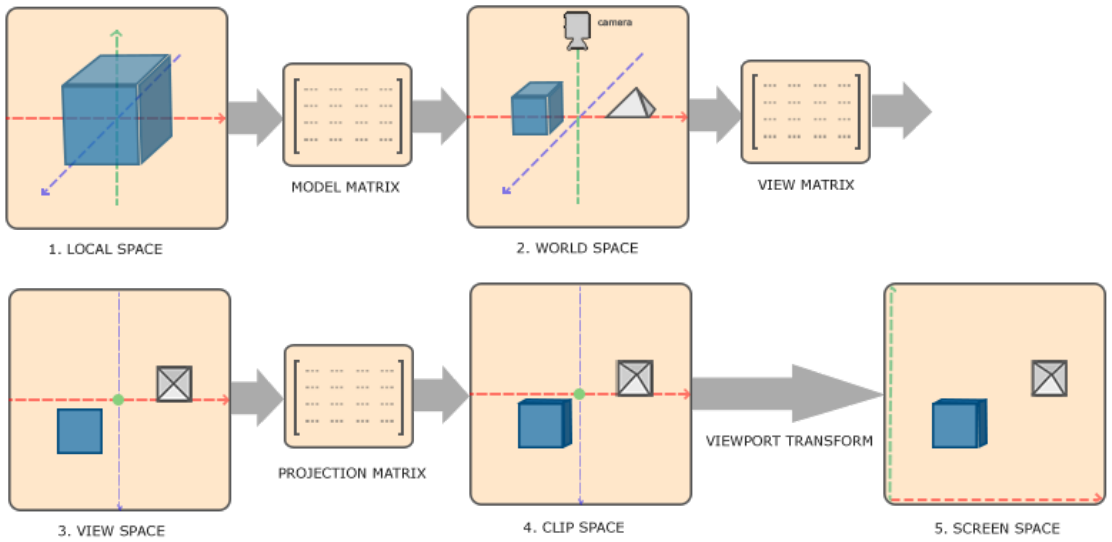
```
public class OpenedArgs : GLib.SignalArgs
{
  public GLib.IFile Files
  { /* ~~^~~ Aquí debería ser GLib.IFile[] */
    get
    {
      return GLib.FileAdapter.GetObject (Args [0] as GLib.Object);
    }
  }

  public int NFiles
  {
    get
    {
      return (int) Args [1];
    }
  }
```

```
    public string Hint
    {
      get
      {
        return (string) Args [2];
      }
    }
  }
```

- `Frontend.Game` : Aquí esta el código del juego como tal
  - `Backend` : Esta clase es el controlador lógico (del **backend**). Mantiene una cola (concurrente) para guardar los eventos del juego según son interpretados. Internamente ejecuta el backend en un proceso aparte y toma de la salida estándar todo lo necesario para representar una partida de dominó, lanzando eventos que describen a cada uno según el controlador gráfico lo requiera.
  - `Window` : Es el controlador gráfico. Usa una ventana proporcionada por la biblioteca **Gtk** para crear una contexto de **OpenGL** y representar el juego usando el motor gráfico `Frontend.Engine`
- `Frontend.Game.Objects` : Colección de objetos gráficos
  - `AtrilObject` : Representa las fichas del jugador de turno. Actualmente en desarrollo
  - `PieceBoard` : Representa el tablero sobre el cual se muestran las fichas. Ademas implementa la lógica para posicionar correctamente las fichas según se juegan, y desde luego mantiene una lista interna de objetos `PieceObject` que representan las fichas (solo están soportadas las fichas con dos caras)
  - `PieceModel` : Encapsula el modulo y las textures de una ficha concreta. También se encarga de crear la texture correspondiente a su número procesalmente (usa la biblioteca **Cairo**, parte del paquete de **GtkSharp** para generarlas), por lo que el valor de las caras de una ficha puede ser totalmente arbitraria
  - `PieceObject` : implementa una sencilla lógica de lista (referencias next, prev, etc.) que se utiliza en la clase `PieceBoard` para dibujar cada una; también facilita el trabajo ya que simula las conexiones de las fichas en un tablero real
- `Frontend.Engine` : Aquí es donde más trabajo puse en comparación con el resto del proyecto, ya que es de muy bajo nivel (aquí hay trabajo con vertices, vectores y matrices de transformaciones)
  - `Gl` : Representa un contexto del motor gráfico. Asume que se ejecuta de forma exclusiva en un contexto de **OpenGL**, aunque no asume control directo sobre el renderizado (deja libertad al usuario para renderizar cada cuadro cuando quiera y configurar el `Viewport` según convenga). Como tal, maneja la iniciación del contexto de **OpenGL** (usa la biblioteca **OpenTK** para el acceso de "alto nivel" a la

API de **OpenGL**). Todo lo relacionado con el motor gráfico se accede por aquí

- `Gl.Camera` : encapsula las matrices de transformación y las abstracciones necesaria para manipularlas. En concreto guarda las matrices de proyección y de cámara, porque la matriz del modelo las proporciona cada `Engine.Object` . El proceso empieza con las coordenadas en espacio local (local space), estas se multiplican por la matriz de model, que transforma en espacio global; se multiplica por la matriz de cámara (world space), que las transforma en espacio de cámara (view space); se multiplica por la matriz de proyección que las transforma en recortado (clipping space); al final **OpenGL** internamente las transforma a espacio de pantalla (screen space) usando la configuración del viewport



- `Gl.Dds` : implementa un cargador para el formato DDS (DirectDraw Surface) que es un formato de imagen comprimido que puede ser usado directamente por **OpenGL**, o sea, no necesita descomprimirse en datas de pixeles para que pueda usarse en una textura
- `Gl.DirLight` : Encapsula los datos de una luz direccional sin posición, algo así como una luz ambiental que tiene una dirección definida; por ejemplo la luz del Sol incide sobre todas partes, pero desde una dirección definida
- `Gl.IBindable` : Representa un objeto de **OpenGL** que debe "bindearse" para usarlo desde un shader. Actualmente solo lo implementan las clases `Gl.Ssbo` y `Gl.Ubo` y no considero que en el futuro lo usen más clases
- `Gl.IDrawable` : Representa un objeto que puede dibujarse
- `Gl.ILocalizable` : Representa un objeto que tiene una posición definida en espacio global (world space)
- `Gl.IPackable` : Representa un objeto que puede empaquetar sus datos en un flujo de bits. Usado por las las implementaciones de `Gl.Light` para empaquetarse en un *SSBO*
- `Gl.IRotable` : Representa un objeto al que se puede rotar usando una vector

como guía

- `Gl.IScalable` : Representa un objeto al que puede escalarse
- `Gl.Light` : clase abstracta base para las luces que pueden adicionarse en una escena. Como tal almacena los valor cromáticos de la luz
- `Gl.Loader` : clase estática que se encarga de cargar en tiempo de ejecución la API de **OpenGL**. Es necesaria porque **OpenGL** es solamente un standard y cada fabricante de tarjetas gráficas crea su propia implementación (**OpenGL** es algo así como una interfaz que cada productor implementa) y que difieren mucho entre cada computadora. Esta clase usa la API especifica para cada plataforma para cargar la biblioteca correspondiente y obtener las funciones (y extensiones) que cada cual implementa
- `Gl.Material` : encapsula un material, esto es, un concepto abstracto de material. Cada material tiene mapas de iluminación (comúnmente las más reconocibles como texturas), de normales, la brillantez, la rugosidad, etc.
- `Gl.Model` : clase base que representa un modelo. Un modelo es un conjunto de "meshes", que a su vez son un conjunto de vertices, y los datos asociados a estos
- `Gl.Pencil` : esta clase implementa en concepto abstracto de lápiz. En este motor gráfico el lápiz es lo que se utiliza para dibujar las primitivas gráficas (triángulos) de cada modelo. Principalmente ayuda al rendimiento porque disminuye la cantidad de cambios de contexto necesarios para dibujar una escena. En **OpenGL** existen unos objetos llamados *array de vertices* (vertex array object) o VAO, que representan la estructura de una buffer de vertices; cambiar de VAO es generalmente costoso en tiempo, por lo que lo optimo es disponer de un VAO que se utiliza para dibujar toda la escena

```
/* Pencil layout */
layout (location = 0) in vec3 vPos;
layout (location = 1) in vec3 vNormal;
layout (location = 2) in vec3 vTexCoords;
layout (location = 3) in vec3 vTangent;
layout (location = 4) in vec3 vBitangent;

/* ~~^~~ Esto es un VAO como puede usarse desde un shader */
/* Cada "location" es un atributo que puede activarse en  */
/* el VAO y corresponde a una sección de un elemento del  */
/* array de vertices                                      */
```

- `Gl.PointLight` : clase que guarda los datos de una luz omnidireccional con una posición concreta (como una farola)
- `Gl.Program` : clase que encapsula un programa de shaders. Cada programa se compone de varios shaders, que juntos se envían a la GPU del ordenador y se

ejecutan cada vez que se dibuja algo

- `Gl.SingleModel` : implementa un modelo básico
- `Gl.SpotLight` : clase que guarda los datos de una luz direccional con una posición concreta (como una linterna)
- `Gl.Ssbo` : encapsula un SSBO (Shader Storage Buffer Object), que es un buffer independiente del programa (`Gl.Program`) que puede accederse desde un shader. Posee la cualidad de ser re-dimensionable y modificable (desde el shader), por lo cual puede usarse como una `IList<T>` para almacenar objectos que implementen la interfaz `Gl.IPackable` y puedan ser accesibles desde el código del shader. Actualmente `Gl` usa tres de estos para almacenar los tres tipos de luces que están implementadas
- `Gl.Ubo` : encapsula un UBO (Uniform buffer object), que es una buffer independiente del programa (`Gl.Program`) que puede accederse desde un shader. Es similar al SSBO, pero es de solo lectura (desde el shader) y no se puede re-dimensionar una vez creado, pero es más rápido. Actualmente `Gl` usa uno para almacenar las matrices de transformación
- `Object` : es la base para los objetos representables con el motor gráfico
- `Skybox` : es una objecto especial que representa un "skybox", que no es mas que un modelo muy grande que contiene toda la escena y provoca la ilusión de un espacio abierto