ChatGPT

Rediseño de nodos de entrada/salida en File-Nodes

Estado actual: Group Input vinculado a la escena activa

En la implementación actual del addon, el nodo **Group Input** funciona como la entrada genérica del grupo de nodos, exponiendo los parámetros definidos en la interfaz del `FileNodesTree`. Por defecto, este nodo incluye un **socket** de tipo Scene llamado "Scene" que siempre se resuelve a la escena activa del contexto (`bpy.context.scene`). Esto se puede ver en el método `process` del `FNGroupInputNode`: si el nombre del ítem de interfaz es "Scene", el nodo retorna la escena activa del contexto; de lo contrario, toma el valor desde el modificador activo (`mod`) usando `mod.get_input_value` [1]. En otras palabras, **Group Input** tiene una dependencia fija de la escena global actual. Además, al crear un nuevo **File Node Modifier**, el código agrega por defecto un socket de entrada "Scene" (tipo `FNSocketScene`) y uno de salida "Scene" en la interfaz del nodo grupo [2], e inserta automáticamente un nodo **Group Input** y un **Group Output** conectados entre sí mediante ese socket de escena [3]. Esto significa que inicialmente el árbol de nodos siempre espera una escena de entrada (tomada del contexto) y la pasa a la salida sin modificaciones. El **Group Output** actualmente es un nodo requerido que define las salidas del grupo, aunque su lógica es mínima (su `process` no hace nada) [4] y sirve solo como punto terminal para los sockets de salida definidos en la interfaz.

Problemas detectados: Esta arquitectura acopla el nodo Group Input a `bpy.context.scene`, impidiendo usar el árbol de nodos con escenas distintas a la activa sin intervención manual. El valor "Scene" del Group Input no puede ser configurado por el usuario (la UI del modificador permite seleccionar una escena distinta en la lista de inputs, pero el código la ignora y sigue usando la activa) [5]. Esto limita la flexibilidad y reutilización del nodo en diferentes contextos. Asimismo, exigir siempre un Group Output con una escena de salida introduce complejidad innecesaria si se desea tener diferentes tipos de salidas finales o incluso ningún resultado persistente.

Nodo UI Input independiente de la escena activa

Se propone rediseñar el nodo Group Input para convertirlo en un nodo **"UI Input"** (entrada de interfaz de usuario) totalmente independiente de la escena activa. Este nodo seguirá sirviendo como panel visual de entrada para configurar parámetros del árbol de nodos (tal como hasta ahora lo hacía Group Input), pero **sin depender de** `bpy.context.scene` **ni requerir una Scene como input obligatoria**. Los cambios sugeridos son:

- **Eliminar la dependencia de la escena activa en** `FNGroupInputNode.process`**:** En el código fuente, esto implica quitar la condición especial que verifica `if item.name == "Scene": outputs[item.name] = context.scene` [5]. En su lugar, **si el interfaz del grupo define un socket "Scene"**, el nodo debería tratarlo como cualquier otro parámetro: obtener su valor a través del modificador activo (`mod.get_input_value("Scene")`) o retornar `None` si no se ha especificado ninguno. De esta manera, la escena de entrada será la que el usuario haya seleccionado explícitamente en la UI del modificador (propiedad `scene_value` del `FileNodeModInput`) y ya no siempre la escena global activa. *Esto permite desacoplar el nodo de la escena del contexto.*

- **Entrada de escena opcional:** Con el rediseño, el nodo UI Input no debería asumir que siempre habrá una escena como parámetro. Se podría **dejar de crear por defecto el socket "Scene" en la interfaz** al añadir un nuevo modificador. Actualmente, la operación `FN_OT_mod_add` agrega siempre `iface.new_socket(name="Scene", in_out='INPUT', socket_type='FNSocketScene')` [2]. Se recomienda eliminar estas líneas para que un nuevo árbol File Nodes empiece sin entradas predefinidas (o al menos que no inserte automáticamente la escena). El nodo **Group Input** (o **UI Input**) inicial aún se puede crear vacío para servir de panel, pero sin sockets ligados a la escena a menos que el usuario los agregue manualmente. En caso de querer mantener una escena por defecto para facilidad, una alternativa es asignar la escena activa inicial **una sola vez** al crear el modificador (por ejemplo, estableciendo `mod.inputs["Scene"].scene_value = context.scene` tras sincronizar la interfaz) en lugar de evaluarla dinámicamente en cada `process`. Sin embargo, es más consistente dejar que el usuario decida la escena de entrada.

- **Conservación de la interfaz de grupo:** Este nodo UI Input seguirá sincronizando sus sockets de salida con los ítems de entrada definidos en `tree.interface`. La clase `FNGroupInputNode` ya implementa `_sync_with_interface` para añadir/quitar sockets según la definición de la interfaz [6] [7]. Esto debe mantenerse. La idea es que el **panel de entrada visual** muestre todos los parámetros configurables del árbol (float, int, objetos, escenas, etc.), tal como hace Group Input actualmente, pero ahora cada valor vendrá de las propiedades del modificador (definidas en `FileNodeModItem.inputs`) sin excepciones. Así, el usuario puede usar el panel de propiedades (sección *File Nodes* de la Scene) para ajustar estos valores [8], o incluso se podría extender la interfaz para permitir editarlos directamente sobre el nodo en el editor de nodos (por ejemplo, usando `draw_buttons` para mostrar controles en el propio nodo, similar a los nodos de entrada constante como *Float Input* [9] [10]). Esto último sería un plus opcional para mejorar la usabilidad, aunque no estrictamente necesario ya que la configuración vía el modificador funciona.

Implementando estos cambios, el **nuevo nodo UI Input** dejará de depender de `bpy.context.scene`. Como resultado, se podrá seleccionar explícitamente cualquier escena (u otros valores) como entrada desde la UI, y esa será la que el nodo entregue al resto del árbol. Técnicamente, habría que modificar la clase `FNGroupInputNode` en *nodes/group_input.py* (particularmente su método `process`) y ajustar la lógica de creación por defecto en *modifiers.py* (clase `FN_OT_mod_add`) para no añadir la escena por omisión. Estos cambios aseguran que el árbol de nodos sea más autónomo y reutilizable, y que **no requiera una Scene activa ni un contexto global específico para funcionar correctamente**.

Eliminación del Group Output obligatorio en la salida

Actualmente, cada árbol File Nodes incluye un nodo **Group Output** al final conectado a la salida "Scene" de la interfaz [2]. Este nodo actúa como contenedor de los sockets de salida definidos en la interfaz del nodo grupo, pero impone que siempre exista al menos un output (la escena). Para flexibilizar el sistema, se propone **remover la lógica que hace a Group Output un nodo obligatorio** y adoptar en su lugar nodos de salida explícitos según el tipo de resultado que se desee (p. ej. escenas a crear, renders a ejecutar, etc., que detallaremos más adelante).

Concretamente, las recomendaciones son:

- **Dejar de insertar automáticamente el Group Output y la salida "Scene":** En la función que inicializa un nuevo FileNodesTree (la operación *Add File Node Modifier*), eliminar la creación de

`iface.new_socket(..., in_out='OUTPUT', socket_type='FNSocketScene')` y la adición del nodo `FNGroupOutputNode` [2]. De este modo, un nuevo árbol no contendrá salidas hasta que el usuario agregue nodos de salida específicos manualmente. La clase `FNGroupOutputNode` podría eventualmente ser deprecada o eliminada del registro si deja de usarse. Al no tener Group Output, también se sugiere deshabilitar el uso de la **interfaz de grupo para outputs** (`bl_use_group_interface`) o simplemente ignorarla, ya que los nuevos nodos terminales manejarán las salidas de forma explícita.

- **Verificar la evaluación sin Group Output:** Afortunadamente, el motor de evaluación actual no depende de un nodo de salida particular. El código de `evaluate_tree` recorre **todos los nodos del árbol y evalúa cada uno** indiscriminadamente [11], registrando sus salidas en un diccionario interno, pero no utiliza un "resultado final" del Group Output. Por tanto, remover el Group Output **no rompe la lógica de evaluación**, pues los nodos igualmente ejecutarán sus métodos `process` durante la evaluación global. En otras palabras, el sistema ya evalúa todos los nodos independientemente de cómo estén conectados, de modo que la ausencia de un nodo de salida no impedirá que se ejecuten las operaciones en el árbol (aunque sería recomendable en un futuro optimizar evaluando solo lo necesario, p. ej. rastreando nodos terminales verdaderos).

- **Outputs actuales manejados implícitamente:** Hasta ahora, cualquier efecto producido por el árbol (crear o modificar datos de Blender) ocurría durante la evaluación y, si correspondía, podía revertirse al finalizar según lo registrado en `mod.store_original` / `reset_to_originals`. Por ejemplo, un nodo como *Set World to Scene* guarda el world original y asigna otro temporalmente [12], y luego esa asignación se revierte al resetear [13]. Un nodo *New Scene*, sin embargo, **crea** una nueva escena en `bpy.data` inmediatamente [14]. Actualmente no existe un mecanismo para eliminar esas escenas si no se usan, por lo que permanecen en el .blend. Este comportamiento refuerza la necesidad de introducir nodos de salida dedicados: en lugar de depender de Group Output (que no ejecuta nada) para determinar qué cambios son definitivos, se tendrán nodos explícitos que realicen la acción final (renderizar, guardar escenas, etc.).

En resumen, remover el Group Output obligatorio simplificará el flujo: el árbol ya no tendrá un nodo de salida genérico fijo, sino que el usuario decidirá qué **nodo(s) de salida explícitos** colocar en función de lo que quiera lograr. Esto limpiará la interfaz (por ejemplo, no siempre habrá una "Scene" de salida a la fuerza) y sentará las bases para manejar las salidas de manera más controlada.

Nuevos nodos de salida explícitos

Para sustituir la función del antiguo Group Output, se propone implementar al menos dos nuevos tipos de **nodos terminales** en el sistema:

Nodo "Render Scenes" (Renderizar Escenas en batch)

Este nodo estará diseñado para tomar una o varias escenas como entrada y proporcionar una forma de **renderizado por lotes** (batch) desde la interfaz de usuario. Sus características y consideraciones técnicas:

- **Entradas:** Tendrá un socket de entrada del tipo **Scene List** (`FNSocketSceneList`) o similar, que pueda recibir **una lista de escenas**. Esto permitirá encadenar múltiples escenas (p. ej. provenientes de varias ramas del nodo o de un *Create List* node combinando escenas [15]) para renderizarlas todas de forma secuencial. Alternativamente, se podría soportar uno o varios

sockets de escena individuales, pero usar un *list socket* es más escalable y consistente con los tipos ya definidos en el addon.

- **Salidas:** Es un nodo terminal, por lo que no necesita sockets de salida para continuar el flujo de datos. Su propósito es colateral: generar imágenes. Podría no retornar nada (o potencialmente un booleano de éxito o lista de rutas renderizadas si se quisiera informar algo, aunque esto no es imprescindible y complicaría la evaluación).

- **Ejecución controlada por UI: Este nodo no debe iniciar renderizados durante la evaluación automática normal.** En su método `process` probablemente no hará nada (o simplemente pasará las escenas de entrada tal cual) para no disparar render inadvertidamente. La idea es exponer un **botón de "Render"** en la interfaz del nodo (en el Node Editor) o en el panel del modificador, que el usuario pulsará manualmente. Para lograrlo, se puede sobrescribir `draw_buttons` en la clase del nodo para añadir un botón UI. Por ejemplo, `layout.operator("file_nodes.render_scenes", text="Render Scenes")`, donde `file_nodes.render_scenes` sería un nuevo operador personalizado.

- **Operador de renderizado batch:** Se debe implementar un operador (`Operator`) asociado al nodo *Render Scenes*. Este operador, al ejecutarse (al pulsar el botón), realizará los pasos necesarios:

- **Evaluar el árbol de nodos** asegurándose de que las escenas de entrada estén actualizadas según las operaciones anteriores del nodo (posiblemente forzando una última evaluación si hubo cambios no auto-evaluados). Dado que el sistema de modificadores suele autoevaluar al cambiar un valor (según `auto_evaluate_if_enabled`), es probable que el árbol ya esté evaluado al momento de pulsar el botón. Aun así, por seguridad, el operador podría llamar internamente a la función `evaluate_tree(context)` antes de renderizar. Esto aplicaría todas las modificaciones temporales necesarias a las escenas (por ejemplo, ajustes de cámara, mundo, etc., que los nodos previos hayan efectuado).
- **Recuperar la lista de escenas a renderizar:** El operador debe obtener las escenas resultantes del socket de entrada. Puesto que durante la evaluación las referencias a escenas ya existen (e.g. escenas nuevas creadas están en `bpy.data.scenes`), se pueden recoger directamente. Una forma es acceder al nodo desde el operador (por ejemplo, usando `context.node` si el botón se pulsó en contexto de ese nodo, o pasar el identificador del nodo al operador). Luego iterar sobre `node.inputs[...].links` y obtener cada escena conectada. Si el socket es de lista (`FNSocketSceneList`), internamente ese valor podría ser una lista Python de `Scene` que se formó en evaluación; podría ser necesario reconstruirla manualmente: por ejemplo, si la entrada viene de un nodo *Create List*, ese nodo en su `process` retorna la lista de escenas [15].
  Implementación sugerida: reutilizar la lógica de evaluación para obtener el valor: se podría llamar a la función recursiva de evaluación sobre el socket de entrada del Render node, similar a `eval_socket`, para obtener la lista de escenas. Otra opción más simple es aprovechar que después de `evaluate_tree`, los sockets de entrada no vinculados cargan su propiedad `.value` (aunque para listas no hay propiedad directa) o que los nodos previos ya crearon las escenas en datos de Blender. En todo caso, el operador debe identificar claramente qué escenas el usuario conectó.
- **Configurar y lanzar render por escena:** Por cada escena obtenida, el operador puede establecer esa escena como la activa y llamar a `bpy.ops.render.render()` o usar `bpy.ops.render.render(animation=False, scene=scene.name)` para renderizar sin necesidad de cambiar contexto. Idealmente, respetará la configuración de render de cada escena (resolución, cámara, engine, etc.), que de hecho pueden haber sido ajustadas dentro del

mismo nodo tree (ej. nodos *Set Render Engine*, *Output Properties*, etc.). Se iterará una por una, iniciando el render secuencialmente. **Nota:** Si se desea render en segundo plano o no bloquear la UI, podría ejecutarse de forma no modal, pero eso es un detalle de implementación Blender (posiblemente aceptable hacerlo simple y bloquear hasta terminar cada render).

- **Finalizar y revertir cambios temporales:** Después de renderizar, habría que considerar si revertir las modificaciones no permanentes. Dado que el sistema de File-Nodes es no destructivo por diseño, tras el render podría ser deseable restablecer las escenas a su estado original (para no dejar, por ejemplo, la iluminación cambiada). Actualmente, las modificaciones temporales se mantienen aplicadas hasta la próxima evaluación con reset. El operador de render podría opcionalmente forzar un reset manual después de completar (por ejemplo, llamando a `mod.reset_to_originals()` para el modificador activo). Sin embargo, esto debe manejarse con cuidado: si el mod permanece *enabled*, un reset inmediato revertiría cambios pero en cuanto el usuario haga otra acción, auto-evaluaría de nuevo y volvería a aplicar modificaciones. Quizá sea más simple **no hacer nada especial** y dejar que el flujo normal (o el usuario deshabilitando el mod) se encargue de la restauración. En todo caso, es importante documentar este comportamiento para que el usuario entienda que el render se hizo con ajustes temporales que luego pueden revertirse.

- **Integración en el addon:** Se necesitará crear la clase de nodo, por ejemplo `class FNRenderScenesNode(Node, FNBaseNode)`, con su correspondiente registro. Debe agregarse a la categoría de nodos (posiblemente en `FILE_NODES_SCENE` en *menu.py*). Además, definir el operador `FN_OT_render_scenes` (en *operators.py* o en un módulo nuevo) con `bl_idname = "file_nodes.render_scenes"`. Este operador debería obtener referencia al nodo activo (vía `context.active_node` asumiendo que es de tipo Render Scenes) o recibir un identificador del nodo. Una vez implementado, el nodo *Render Scenes* aparecerá en el editor de nodos y permitirá al usuario, mediante su botón, ejecutar el render batch sobre las escenas conectadas.

En resumen, el **Render Scenes node** proporcionará una forma segura y manual de lanzar renderizados de las escenas procesadas por el árbol, sin que ello ocurra automáticamente en cada evaluación. Esto evita ejecuciones de render no deseadas, al tiempo que integra la funcionalidad de batch render en el mismo contexto de los File Nodes.

Nodo "Output Scenes" (Crear/actualizar escenas en el .blend)

El nodo **Output Scenes** estará enfocado en tomar escenas generadas o modificadas dentro del node tree y **materializarlas definitivamente en el archivo .blend** una vez completada la evaluación. Esto cubre casos como: crear nuevas escenas, duplicar o modificar escenas de forma no destructiva durante la evaluación y luego decidir persistir esos cambios. Aspectos clave del diseño:

- **Entrada:** Un socket de tipo **Scene** o **Scene List** (similar al Render node) que reciba la(s) escena(s) a consolidar. Usar un *Scene List* permitiría salida múltiple en batch (por ejemplo, varias escenas nuevas creadas en el árbol podrían conectarse todas a un solo Output Scenes node). Cada escena en la lista será procesada para su *commit* final.

- **Sin salida adicional:** Este nodo es terminal, no necesita propagar datos más adelante. Su `process` puede incluso retornar vacío `{}` o simplemente pasar las escenas si se quisiera encadenar (aunque conceptualmente, es fin de línea). El punto importante es lo que hace tras la evaluación.

- **Comportamiento en evaluación:** Durante la evaluación normal, el **Output Scenes** no debería por sí mismo duplicar escenas ni hacer cambios; más bien, actuará como marcador de que ciertas escenas deben conservarse. Una estrategia es que su método `process` simplemente entregue las escenas de entrada tal cual (como paso final) y registre internamente en el modificador activo que esas escenas están marcadas para salida. Dado que en la evaluación todo ocurre con el modificador `mod` en contexto (`_active_mod_item`), el nodo puede acceder a este mediante `get_active_mod_item()` [16]. Por ejemplo, `process` podría hacer:

```
mod = get_active_mod_item()
scenes = inputs.get("Scenes") or []
if mod:
    mod.scenes_to_keep = list(scenes)
return {}
```

(Aquí `scenes_to_keep` sería una nueva propiedad o estructura en `FileNodeModItem` para almacenar las referencias a las escenas que no deben revertirse). De esta forma, el nodo indica qué escenas deben considerarse "output real".

- **No revertir cambios ni eliminar escenas marcadas:** El **mayor desafío técnico** es coordinar con el sistema de revertir cambios (`reset_to_originals`) para que no se deshagan las modificaciones de las escenas de salida. Actualmente, `FileNodeModItem.reset_to_originals()` recorre todo lo almacenado en `_original_values` y revierte propiedades u enlaces modificados [17] [13]. También limpia objetos o colecciones vinculados dinámicamente. Sin embargo, **no maneja la creación de nuevos ID** (escenas, objetos, etc.), lo cual significa que escenas nuevas permanecen a menos que se eliminen manualmente. Con la introducción de Output Scenes, se recomienda actualizar esta lógica:

- **Seguimiento de nuevas escenas creadas:** Cuando un nodo *New Scene* crea una escena (vía `bpy.data.scenes.new` [14]), podría invocar una función del modificador para registrarla como "escena creada temporal". Por ejemplo, agregar en `FileNodeModItem` un método `remember_created_scene(scene)` que almacene la referencia (quizá en `self._original_values["new_scenes"]` lista). Actualmente no existe, pero es simple de añadir. Ninguno de los nodos actuales hace este registro, por lo que habría que modificar *FNNewScene.process* para llamarlo.
- **Modificar `reset_to_originals` para manejar nuevas escenas:** Antes de revertir propiedades, debería eliminar las escenas que se crearon durante la evaluación y que **no** estén marcadas para conservar. Es decir, recorrer `self._original_values.get("new_scenes", [])` y para cada escena que aún exista en `bpy.data.scenes`:
  - Si la escena está en la lista de *scenes_to_keep* (marcada por Output Scenes), se omite su eliminación (queremos conservarla).
  - Si **no** está marcada, significa que era temporal y no deseada, entonces se puede eliminar con `bpy.data.scenes.remove(scene)`. Esto limpiará esa escena del archivo .blend.
- **No revertir propiedades en escenas de salida:** Similarmente, al revertir propiedades guardadas, si alguna corresponde a una escena (u objeto dentro de esa escena) que está marcada para output, deberíamos saltar ese cambio. Por ejemplo, en `reset_to_originals`, donde se hace `setattr(data, attr, value)` para restaurar un valor [13], habría que comprobar si `data` pertenece a una escena de la lista *scenes_to_keep* y en tal caso **no restaurarla** (dejamos los cambios hechos). Esto asegura que, por ejemplo, si se cambió el world

de la escena con *Set World* o se modificaron propiedades de render, esos cambios permanezcan aplicados en la escena final.

- **Finalización de evaluación con commit:** Con las adaptaciones anteriores, la **evaluación final** del árbol resultará en que las escenas conectadas al nodo Output Scenes queden en el estado modificado (o recién creadas) dentro del .blend de forma permanente. Los datos que no estaban marcados serán revertidos o eliminados, cumpliendo la promesa de no-destructividad para todo lo demás. Hay distintas formas de activar este commit:

- Si la evaluación se ejecuta normalmente (por usuario cambiando valores o manual "Evaluate File Nodes"), al terminar, en teoría el mod sigue habilitado y las escenas marcadas siguen presentes con cambios (porque no se revirtieron). Si el usuario guarda el archivo `.blend`, ya contendrá esas escenas.
- Una práctica segura tras finalizar sería **deshabilitar o quitar el modificador**, ya que una vez cumplido su cometido (crear las nuevas escenas), podría no hacer falta seguir evaluando. Al remover el File Node Modifier, se llama internamente a `restore_and_clear()` que actualmente revierte todo y limpia registros [18]. Habría que asegurarse de adaptar eso también para no borrar las escenas de salida. Quizá `restore_and_clear` no debería revertir nada si hay escenas de salida (o al menos omitir esas partes), pero en principio si `reset_to_originals` fue adecuado, *restore_and_clear* puede permanecer principalmente para limpiar la estructura de almacenados.

- Se podría implementar un mecanismo para que, tras una evaluación con Output Scenes, el modificador se autodesactive. No es estrictamente necesario, pero previene que cambios posteriores intenten reevaluar lo ya aplicado. Esto puede ser tan sencillo como que `FNOutputScenesNode.process` al final haga `if mod: mod.enabled = False` (lo que en la UI equivaldría a apagar el "checkbox" del mod). Aunque habría que tener cuidado, porque cambiar mod.enabled dentro de la evaluación podría triggerear una recursión de evaluación. Quizá es mejor dejar esa acción al usuario.

- **Implementación en código:** Crear la clase `FNOutputScenesNode` en *nodes/* (tal vez en un nuevo archivo o en *scene_nodes.py* si existiese). Registrar y añadir al menú (categoría Scene). Su `process` tomará las escenas de entrada y, como se explicó, marcará en el modificador activo cuáles conservar. La clase `FileNodeModItem` en *modifiers.py* se debe extender con:

- Una propiedad o estructura (`scenes_to_keep: list` por ejemplo, quizás no hace falta ser Property de Blender, puede ser solo atributo Python temporal).
- Métodos `remember_created_scene` y lógica en `reset_to_originals` para manejar nuevas escenas y omitir las marcadas. Las secciones del código relevantes para modificar son donde se manejan los originales: tras las unlink de objetos/colecciones [19], antes de restaurar propiedades, introducir la eliminación de nuevas escenas no deseadas. También condicionar la parte de `setattr` para saltar escenas en *scenes_to_keep*.
- Asegurarse de limpiar la lista `scenes_to_keep` después de su uso (por ejemplo, al final de `reset_to_originals` o en `restore_and_clear`), para no acumular referencias inválidas.

Con el **Output Scenes node**, los usuarios podrán, por ejemplo, crear escenas procedurales (con *New Scene*), configurarles propiedades en el nodo tree (cámara, mundo, etc.), y finalmente conectarlas a *Output Scenes* para que se incorporen definitivamente al proyecto. Igualmente, podrán tomar una escena existente (entrando por UI Input u otro *Scene Input* node), aplicar modificaciones temporales y luego hacerlas permanentes sólo al decidir pasarlas por Output Scenes. Esta separación garantiza que

nada se guarda a menos que el usuario agregue explícitamente ese nodo de salida, manteniendo el flujo controlado.

Referencias a clases y funciones clave para modificar/crear

- `nodes/group_input.py` – **Clase** `FNGroupInputNode`: Modificar el método `process` [1] para quitar el uso de `context.scene` y usar únicamente los valores del modificador (`mod.get_input_value`). Verificar también `_sync_with_interface` para mantener la generación de sockets virtuales etc., pero eso puede permanecer igual.
- `modifiers.py` – **Operador** `FN_OT_mod_add.execute`: En la sección que crea sockets "Scene" de entrada/salida y los nodos Group Input/Output por defecto [2], eliminar estas inserciones. En su lugar, puede crearse el nodo Group Input solo (sin sockets predefinidos) para que sirva de panel vacío, o incluso no añadir ningún nodo (aunque convendría al menos un Group Input vacío para que el usuario visualice dónde configurar entradas). Si se decide aún agregar una entrada Scene por usabilidad, hacerlo sin forzar contexto: es preferible asignar `mod.inputs["Scene"].scene_value` al contexto.scene inicial, en vez de depender de ello en runtime – aunque idealmente no añadir nada por defecto.
- `nodes/group_output.py` – **Clase** `FNGroupOutputNode`: Después de la transición, este nodo ya no será necesario. Se puede quitar del menú de nodos (ver *menu.py* [20]) y eventualmente eliminar su registro. Si se mantienen por compatibilidad, asegurarse de que su presencia no afecte (probablemente ignorarlo).
- `nodes/input_nodes.py` – **(Opcional) Escena por defecto**: Existe un `FNSceneInputNode` que permite al usuario escoger una escena arbitraria dentro del árbol [21]. Con el UI Input node rediseñado, usar este nodo individual es redundante si se expone un parámetro de escena en la interfaz. Aun así, ese nodo sigue útil para casos donde no se usa el modificador interface. No requiere cambios, pero se menciona porque ahora *Group Input* no será la única vía de proveer escenas.
- `modifiers.py` – **Clase** `FileNodeModItem`: Añadir estructuras para *escenas a conservar*. Por ejemplo, `self.scenes_to_keep = []` inicializable cada eval. Incorporar:
- Método `remember_created_scene(scene)` que almacene la escena en una colección interna (e.g. `self._ensure_storage()["new_scenes"]` as a list). Los nodos creadores (New Scene, New Collection, etc.) deberían llamar a este método después de crear un nuevo datablock.
- Lógica en `reset_to_originals`: antes o después de revertir propiedades, iterar sobre escenas nuevas y eliminar las no marcadas. También filtrar la restauración de propiedades para objetos/escenas marcados. *Ejemplo:*

```
new_scenes = storage.get("new_scenes", [])
for scene in new_scenes:
    if scene not in getattr(self, "scenes_to_keep", []):
        try:
            bpy.data.scenes.remove(scene)
        except Exception:
            pass
# luego proceder a resto de reset...
for (ptr, attr), (data, value) in storage.items():
    if isinstance(data, bpy.types.Scene) and data in getattr(self,
"scenes_to_keep", []):
        continue  # no revertir propiedades de escenas guardadas
    try:
```

```
            setattr(data, attr, value)
    ...
```

(El pseudocódigo asume que se guardaron tuplas `(data_pointer, attr)` para propiedades modificadas [13]). De esta forma, las escenas que vamos a mantener no se tocan, ni sus propiedades, mientras que todo lo demás vuelve atrás.

- Limpiar `scenes_to_keep` al final de la evaluación, o en `restore_and_clear`. Por ejemplo, después de realizar reset, hacer `self.scenes_to_keep.clear()` para que en la siguiente evaluación no arrastre referencias antiguas.

- **Nuevo archivo** `nodes/output_nodes.py` (o similar) – Clases `FNRenderScenesNode` y `FNOutputScenesNode`: implementar estas clases con `bl_idname` únicos, sus `init` (definiendo sockets de entrada adecuados) y registrar en la categoría. Por ejemplo:

```
class FNRenderScenesNode(Node, FNBaseNode):
    bl_idname = "FNRenderScenesNode"
    bl_label = "Render Scenes"
    def init(self, context):
        self.inputs.new('FNSocketSceneList', "Scenes")
    def draw_buttons(self, context, layout):
        layout.operator("file_nodes.render_scenes", text="Render
Scenes")
    def process(self, context, inputs):
        # No automatic processing, maybe just pass scenes through or
nothing
        return {}
```

Y

```
class FNOutputScenesNode(Node, FNBaseNode):
    bl_idname = "FNOutputScenesNode"
    bl_label = "Output Scenes"
    def init(self, context):
        self.inputs.new('FNSocketSceneList', "Scenes")
    def process(self, context, inputs):
        scenes = inputs.get("Scenes") or []
        mod = get_active_mod_item()
        if mod:
            # Mark scenes to keep
            mod.scenes_to_keep = [sc for sc in scenes if sc]
        return {}
```

(El código arriba es orientativo; habría que ajustarlo al estilo del addon, verificar tipos (lista de escenas) y evitar que mod.scenes_to_keep sea None, etc.)
Luego, incluir `NodeItem('FNRenderScenesNode')` y `NodeItem('FNOutputScenesNode')` en la lista de nodos de la categoría adecuada del menú (probablemente **Scene**).

- **Nuevo operador** `FN_OT_render_scenes` – En *operators.py*: implementar el operador de render batch con `bl_idname = "file_nodes.render_scenes"` y `execute` que realice los pasos mencionados (evaluar, iterar escenas, llamar renders). Este operador puede localizar al nodo activo vía `context.active_node` asumiendo que el botón solo aparece cuando el nodo está seleccionado. Dentro de `execute`, se puede hacer:

```
node = context.active_node
if isinstance(node, FNRenderScenesNode):
    # Asegurar evaluación actual
    bpy.ops.file_nodes.evaluate()
# llamar el operador de evaluar nodos ya existente
    scenes = []
    # Obtener escenas desde el socket de entrada
    for link in node.inputs['Scenes'].links:
        from_node = link.from_node
        output_name = link.from_socket.name
        # Se puede usar la función interna eval_node del sistema o
similar para obtener el valor...
        # Simplificación: si la escena proviene de un Scene Input o UI
Input, ese nodo ya tiene .value
        # Si proviene de CreateList, necesitamos reconstruir lista:
podríamos simplemente evaluar node.inputs['Scenes'].is_linked... etc.
    # Realizar render por cada escena
    for scene in scenes:
        if scene:
            bpy.context.window.scene = scene
# establecer escena activa (opcional)
            bpy.ops.render.render(write_still=True)  # o
render.animation si se desea
    return {'FINISHED'}
else:
    return {'CANCELLED'}
```

  El pseudocódigo ilustra la idea; la implementación final debe manejar la obtención robusta de la lista de escenas. Podría ser necesario replicar parcialmente la lógica de `_evaluate_tree` para el subárbol, o simplemente confiar en que la evaluación normal ya ejecutó y dejó las escenas creadas/configuradas.

Con todo lo anterior, **File-Nodes** evolucionaría hacia un esquema más modular: un nodo de entrada configurable por UI que no asume contexto global, nodos de salida especializados para acciones finales, y la eliminación de la dependencia rígida en un Group Output genérico. Estas modificaciones harían el sistema más intuitivo (el usuario ve claramente qué entra y qué sale del nodo tree), más seguro (evitando efectos colaterales no deseados), y más alineado con la filosofía de nodos de Blender (similar a Geometry Nodes, donde la interfaz de grupo es independiente y existen nodos específicos para outputs como Viewer, etc.). Cada cambio propuesto se fundamenta en la lectura del código actual – por ejemplo, la necesidad de eliminar la referencia a `context.scene` en **Group Input** [5] y de dejar de generar sockets "Scene" por defecto [2] – y extiende funcionalidades ya esbozadas (la evaluación evalúa todos los nodos [11], los sockets list ya existen para manejar múltiples escenas, las propiedades temporales se almacenan para revertir [12] [13], etc.). En la implementación final, habría que probar cuidadosamente casos como: múltiples Output Scenes en un mismo árbol, combinación de Render

Scenes y Output Scenes, o la interacción con la secuencia de evaluación de varios modificadores en pila, asegurando que cada nodo terminal opere correctamente en su contexto. Con las recomendaciones dadas sobre qué funciones y clases tocar, el desarrollador del addon tendrá una guía clara para realizar estos cambios arquitectónicos en el código.

Fuentes del código relevante:

- Implementación actual de **Group Input** (usa escena activa) [1].
- Creación por defecto de sockets "Scene" y nodos Group Input/Output [2].
- Implementación actual de **Group Output** (nodo terminal sin operación) [4].
- Lógica de evaluación de todos los nodos del árbol (no depende de Group Output) [11].
- Ejemplo de nodo que modifica escena con `store_original` (Set World) [12] y revertido posterior [13].
- Ejemplo de nodo que crea nuevas escenas (New Scene) [14].

---

[1] [5] [6] [7] group_input.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/group_input.py

[2] [3] [13] [17] [18] [19] modifiers.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/modifiers.py

[4] group_output.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/group_output.py

[8] ui.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/ui.py

[9] [10] [21] input_nodes.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/input_nodes.py

[11] operators.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/operators.py

[12] set_world.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/set_world.py

[14] new_scene.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/new_scene.py

[15] create_list.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/create_list.py

[16] output_props.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/nodes/output_props.py

[20] menu.py
https://github.com/pvtrcorps/File-Nodes/blob/061d07eb23367fae52b6bce8cb71260a12950452/menu.py