DIGITAL ARTS & ENTERTAINMENT

Scenegraph & Dirty flag

howest
university of applied sciences

Meltem Ozcelik (Concept: Amandine Gerard) / Daniil Shashenkov (Concept: Tooth Wu) / Joao Desager (Concept: Tan Zhi Hui)

Components and their owner

We saw this a lot... what's wrong with this? (4 things at least)

```cpp
class GameObject;
class BaseComponent
{
  public:
    BaseComponent() = default;
    virtual ~BaseComponent() = default;
    BaseComponent(const BaseComponent& other) = delete;
    BaseComponent(BaseComponent&& other) = delete;
    BaseComponent& operator=(const BaseComponent& other) = delete;
    BaseComponent& operator=(BaseComponent&& other) = delete;

    virtual void Update() = 0;
    virtual void FixedUpdate() = 0;

    void SetOwner(GameObject* pOwner) { m_pOwner = pOwner; };
  protected:
    GameObject* m_pOwner{};
};
```

Components and their owner

Slightly better...

```
class GameObject;
class BaseComponent
{
     GameObject* m_pOwner;
  public:
     BaseComponent(GameObject* pOwner) : m_pOwner(pOwner) {};
     virtual ~BaseComponent() = default;
     BaseComponent(const BaseComponent& other) = delete;
     BaseComponent(BaseComponent&& other) = delete;
     BaseComponent& operator=(const BaseComponent& other) = delete;
     BaseComponent& operator=(BaseComponent&& other) = delete;

     virtual void Update() = 0;
     virtual void FixedUpdate() = 0;

     void SetOwner(GameObject* pOwner) { m_pOwner = pOwner; };
  protected:
     GameObject* GetOwner() const { return m_pOwner; }
};
```

Components and their owner

SetOwner has to do **four** things:

- Check if the new owner is not null
- Remove itself as a component from the previous owner.
- Set the given owner on itself.
- Add itself as a component to the given parent.

```
void SetOwner(GameObject* pOwner)
{
  assert(pOwner);
  if (!pOwner)
    return;
  if (m_pOwner)
    m_pOwner->RemoveComponent(this);
  m_pOwner = pOwner;
  m_pOwner->AddComponent(this);
}
```

Good idea?

Components and their owner

Better

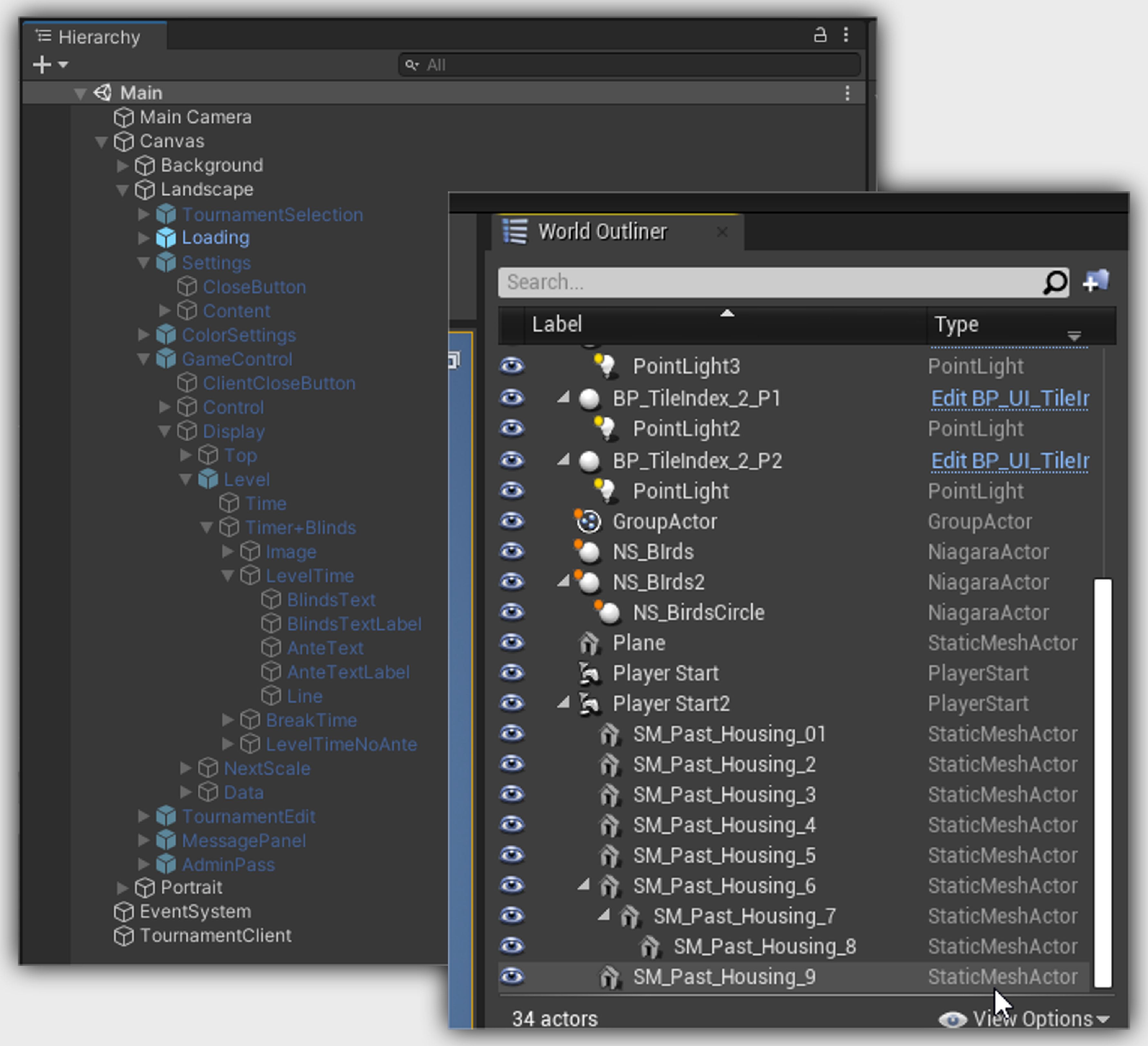
```
class GameObject;
class BaseComponent
{
    GameObject* m_pOwner;
  public:
    virtual ~BaseComponent() = default;
    BaseComponent(const BaseComponent& other) = delete;
    BaseComponent(BaseComponent&& other) = delete;
    BaseComponent& operator=(const BaseComponent& other) = delete;
    BaseComponent& operator=(BaseComponent&& other) = delete;

    virtual void Update() = 0;
    virtual void FixedUpdate() = 0;
  protected:
    explicit BaseComponent(GameObject* pOwner) : m_pOwner(pOwner) {}
    GameObject* GetOwner() const { return m_pOwner; }
};
```

Structure

Well known in engines and 3D modelling software

Hierarchical structure of objects in the scene



- Main
  - Main Camera
  - Canvas
    - Background
    - Landscape
      - TournamentSelection
      - Loading
      - Settings
        - CloseButton
        - Content
      - ColorSettings
      - GameControl
        - ClientCloseButton
        - Control
        - Display
          - Top
          - Level
            - Time
            - Timer+Blinds
              - Image
              - LevelTime
                - BlindsText
                - BlindsTextLabel
                - AnteText
                - AnteTextLabel
                - Line
              - BreakTime
              - LevelTimeNoAnte
          - NextScale
          - Data
      - TournamentEdit
      - MessagePanel
      - AdminPass
    - Portrait
  - EventSystem
  - TournamentClient

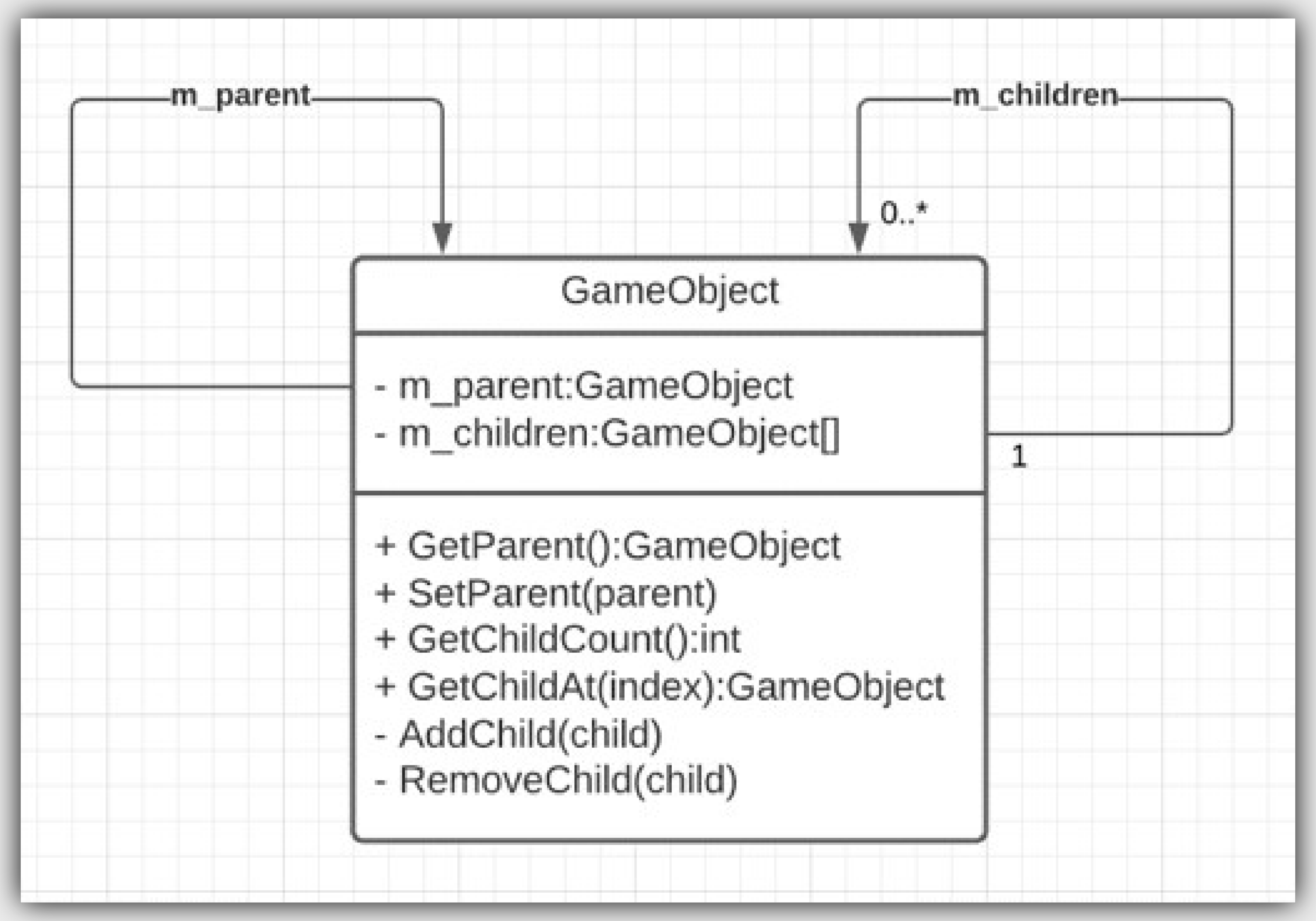
Label	Type
PointLight3	PointLight
BP_TileIndex_2_P1	Edit BP_UI_TileIr
PointLight2	PointLight
BP_TileIndex_2_P2	Edit BP_UI_TileIr
PointLight	PointLight
GroupActor	GroupActor
NS_BIrds	NiagaraActor
NS_BIrds2	NiagaraActor
NS_BirdsCircle	NiagaraActor
Plane	StaticMeshActor
Player Start	PlayerStart
Player Start2	PlayerStart
SM_Past_Housing_01	StaticMeshActor
SM_Past_Housing_2	StaticMeshActor
SM_Past_Housing_3	StaticMeshActor
SM_Past_Housing_4	StaticMeshActor
SM_Past_Housing_5	StaticMeshActor
SM_Past_Housing_6	StaticMeshActor
SM_Past_Housing_7	StaticMeshActor
SM_Past_Housing_8	StaticMeshActor
SM_Past_Housing_9	StaticMeshActor

34 actors   View Options

Structure

Variations are possible, but: each method must ensure that you end up with a correct parent-child relationship when it's done.

Pay close attention to this diagram, it clearly documents what methods you need and how you should implement them.



Implementation

Is this correct?

```
void SetParent(GameObject* parent) { m_parent = parent; }

void GetParent() const { return m_parent; }

size_t GetChildCount() const { return m_children.size(); }

GameObject* GetChildAt(unsigned int index) const { return m_children[index]; }

void RemoveChild(GameObject* child) {
  m_children.erase(std::remove(m_children.begin(), m_children.end(), child), m_children.end());
}

void AddChild(GameObject* child) { m_children.emplace_back(child); }
```

How then?

SetParent has to do **five** things:

- Check if the new parent is valid (not itself or one of its children)
- Remove itself as a child from the previous parent (if any).
- Set the given parent on itself.
- Add itself as a child to the given parent.
- Update position, rotation and scale

How then?

SetParent has to do **five** things:

- Check if the new parent is valid (not itself or one of its children)
- Remove itself from the previous parent (if any).
- Set the given parent on itself.
- Add itself as a child to the given parent.
- Update position, rotation and scale

AddChild has to do **five** things

- Check if the new child is valid (not null and not one of its parents)
- Remove the given child from the child's previous parent
- Set itself as parent of the child
- Add the child to its children list.
- Update position, rotation and scale

How then?

SetParent has to do **five** things:

- Check if the new parent is valid (not itself or one of its children)
- Remove itself from the previous parent (if any).
- Set the given parent on itself.
- Add itself as a child to the given parent.
- Update position, rotation and scale

AddChild has to do **five** things

- Check if the new child is valid (not null and not one of its parents)
- Remove the given child from the child's previous parent
- Set itself as parent of the child
- Add the child to its children list.
- Update position, rotation and scale

RemoveChild has to do **four** things

- Check if the child is valid (not null and one of its children)
- Remove the given child from the children list
- Remove itself as a parent of the child.
- Update position, rotation and scale

How then?

SetParent has to do **five** things:

- Check if the new parent is valid (not itself or one of its children)
- Remove itself from the previous parent (`RemoveChild` ?).
- Set the given parent on itself.
- Add itself as a child to the given parent (`AddChild` ?).
- Update position, rotation and scale

AddChild has to do **five** things

- Check if the new child is valid (not null and not one of its parents)
- Remove the given child from the child's previous parent (`RemoveChild` ?)
- Set itself as parent of the child (`SetParent` ?)
- Add the child to its children list.
- Update position, rotation and scale

RemoveChild has to do **four** things

- Check if the child is valid (not null and one of its children)
- Remove the given child from the children list
- Remove itself as a parent of the child. (`SetParent` ?)
- Update position, rotation and scale

Can SetParent use AddChild to do its job?
Can AddChild use SetParent to do its job?
No – stack overflow would happen

Do we really need AddChild/RemoveChild?
No, being able to set the parent on a GameObject is enough.
Set the parent to nullptr to remove the child from its parent.

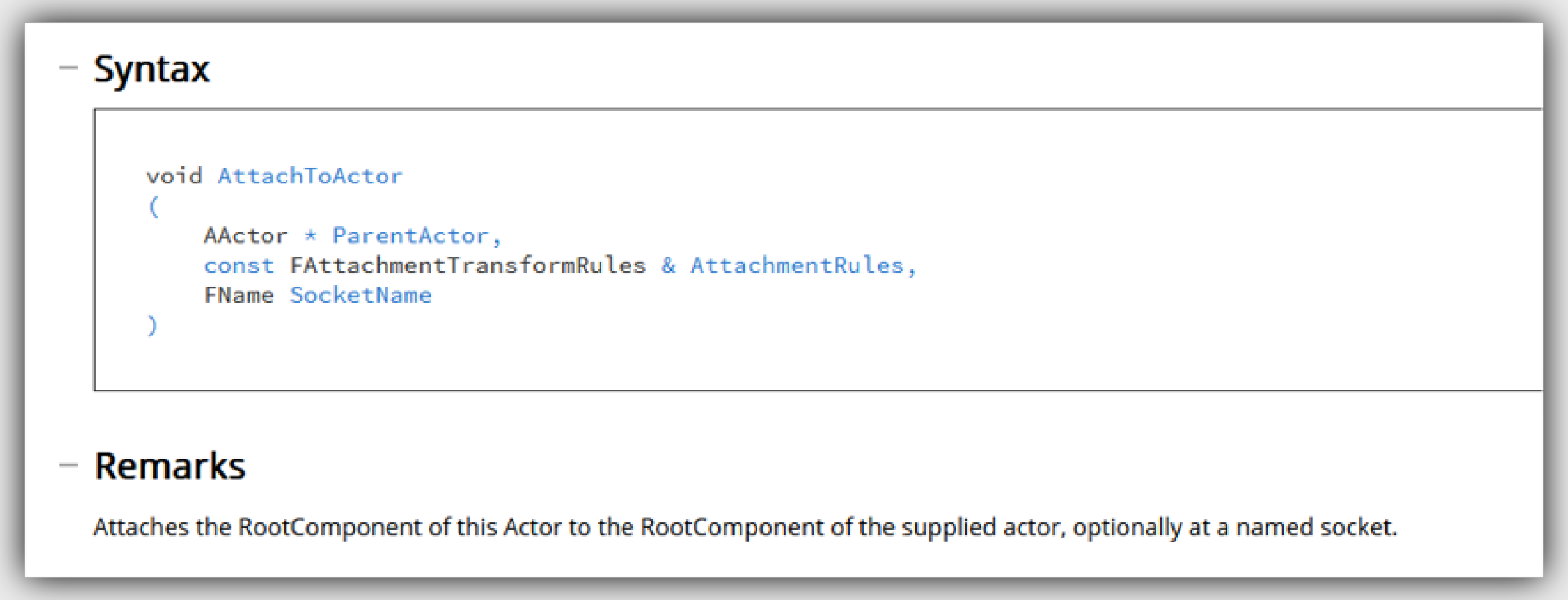
SetParent

When changing the parent, does the child stay at the same place in the world, or not?

SetParent

When changing the parent, does the child stay at the same place in the world, or not?

It depends on the use case, you'll need to provide some options for your users.

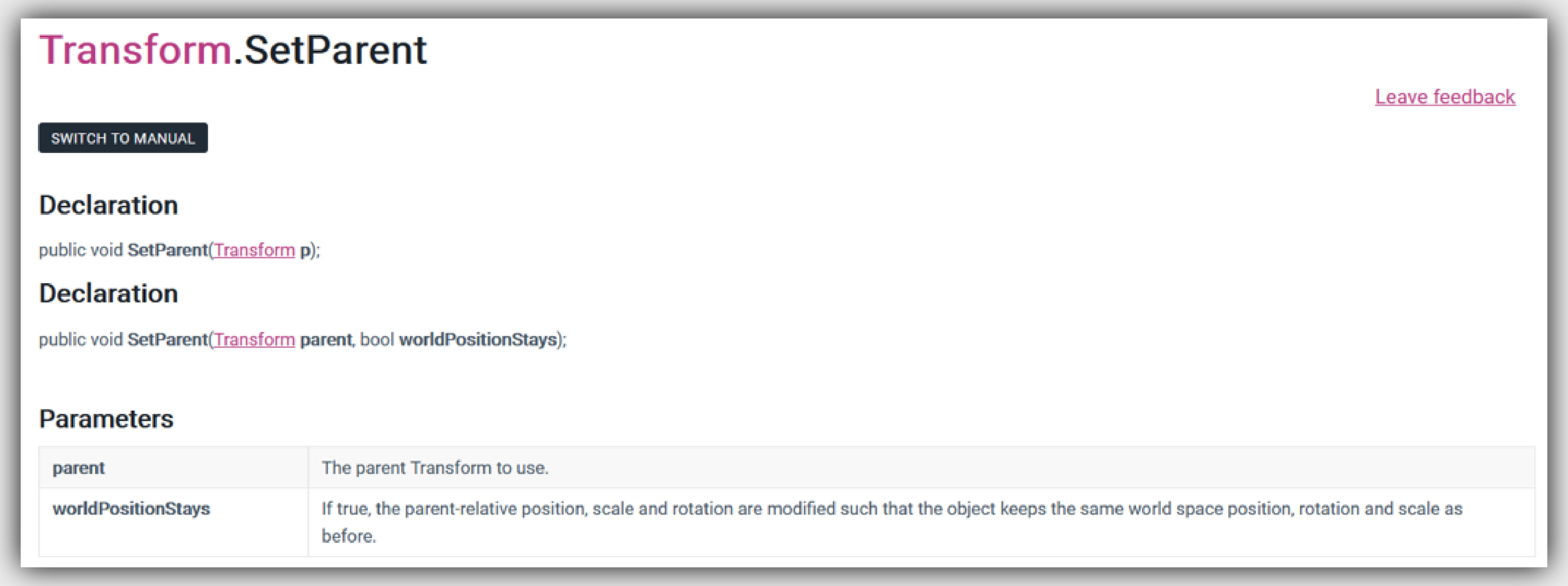


Syntax

```
void AttachToActor
(
    AActor * ParentActor,
    const FAttachmentTransformRules & AttachmentRules,
    FName SocketName
)
```

Remarks

Attaches the RootComponent of this Actor to the RootComponent of the supplied actor, optionally at a named socket.



Transform.SetParent

Leave feedback

SWITCH TO MANUAL

Declaration

public void **SetParent**(Transform **p**);

Declaration

public void **SetParent**(Transform **parent**, bool **worldPositionStays**);

Parameters

parent	The parent Transform to use.
worldPositionStays	If true, the parent-relative position, scale and rotation are modified such that the object keeps the same world space position, rotation and scale as before.

Local vs World space

An objects size, position and rotation in the game world

- Can be expressed in global terms (world space)
- Can be expressed in local terms, relative to the parent (local space)

If no parent, local space == world space

Adding a game object to a parent means either:

- The position in local space changes
- The position in global space changes

When the position of a parent changes, the position of the children change too (in world space).

- When do we calculate the world space transform? Immediately when moving the parent?
- Better: only when we need it.

Local vs World space

Like this?

```
glm::vec3 Transform::GetWorldPosition() const
{
  if(m_parent != nullptr)
    return m_parent->GetWorldPosition() + GetLocalPosition();
  return GetLocalPosition();
}
```

Dirty flag

“A set of **primary data** changes over time. A set of **derived data** is determined from this using some **expensive process**. A **“dirty” flag** tracks when the derived data is out of sync with the primary data. It is **set when the primary data changes**. If the flag is set when the derived data is needed, then **it is reprocessed and the flag is cleared**. Otherwise, the previous **cached derived data** is used.”

What is primary and derived in our context?

Dirty flag

“A set of **primary data** changes over time. A set of **derived data** is determined from this using some **expensive process**. A **“dirty” flag** tracks when the derived data is out of sync with the primary data. It is **set when the primary data changes**. If the flag is set when the derived data is needed, then **it is reprocessed and the flag is cleared**. Otherwise, the previous **cached derived data** is used.”

What is primary and derived in our context?

- Local transform is primary
- World transform is derived

What other example have you seen in Minigin?

Dirty flag

```
void SetParent(GameObject* parent, bool keepWorldPosition)
{
  if(IsChild(parent) || parent == this || m_parent == parent)
    return;
  if (parent == nullptr)
    SetLocalPosition(GetWorldPosition());
  else
  {
    if (keepWorldPosition)
      SetLocalPosition(GetLocalPosition() - parent->GetWorldPosition());
    SetPositionDirty()
    if(m_parent) m_parent->RemoveChild(this);
    m_parent = parent;
    if(m_parent) m_parent->AddChild(this)
}

void SetLocalPosition(const glm::vec3& pos)
{
    m_localPosition = pos;
    SetPositionDirty();
}
```

Dirty flag

```cpp
const glm::vec3& GetWorldPosition() const
{
  if (m_positionIsDirty)
    UpdateWorldPosition();
  return m_worldPosition;
}

void UpdateWorldPosition()
{
  if(m_positionIsDirty)
  {
    if (m_parent == nullptr)
      m_worldPosition = m_localPosition;
    else
      m_worldPosition = m_parent->GetWorldPosition() + m_localPosition;
  }
  m_positionIsDirty = false;
}
```

Considerations

The primary data has to change more often than the derived data is used.

- If you always need the derived data when the primary changes, you can just as well derive it immediately.

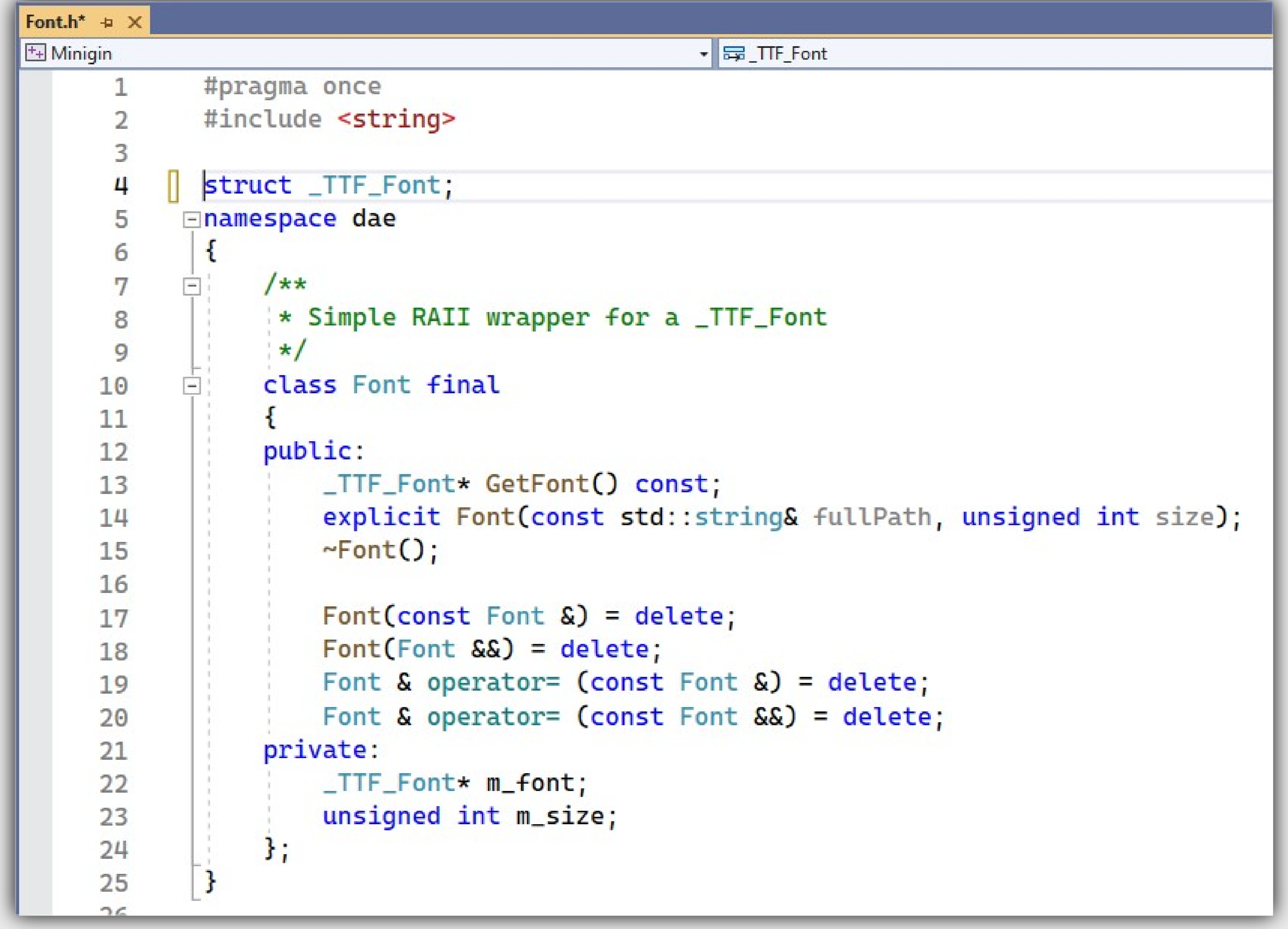
It should be hard/costly to update incrementally.

- Calculating the derived data is a lengthy/costly operation, otherwise just do it.
- Think about the hot/cold path!

When is the flag cleared?

- When the result is needed (as in the example)
- At well-defined checkpoints
- In the background

Find the dirty flag



```cpp
#pragma once
#include <string>

struct _TTF_Font;
namespace dae
{
	/**
	 * Simple RAII wrapper for a _TTF_Font
	 */
	class Font final
	{
	public:
		_TTF_Font* GetFont() const;
		explicit Font(const std::string& fullPath, unsigned int size);
		~Font();

		Font(const Font &) = delete;
		Font(Font &&) = delete;
		Font & operator= (const Font &) = delete;
		Font & operator= (const Font &&) = delete;
	private:
		_TTF_Font* m_font;
		unsigned int m_size;
	};
}
```