GameMindForge Unreal Engine Plugin

The GameMindForge plugin is a powerful tool that allows game developers to integrate GPT-3 language processing and natural language generation into their games. With this plugin, developers can create immersive and engaging gameplay experiences that allow players to interact with NPCs and other characters in a natural and intuitive way.

The plugin features a range of user-friendly tools and functionalities, such as easy-to-use blueprints and customizable settings. Developers can create chatbots and other interactive characters that can respond to a wide range of prompts and queries from players, opening up exciting new possibilities for dialogue and storytelling in games. Here are some of the key features.

- Integrated access to OpenAI's ChatGPT API for generating natural language text.
- Simple API for sending text to the GPT-3 API and receiving generated responses.
- Customizable settings for controlling GPT-3 model selection, maximum token count, and creativity temperature.
- Support for multiple GPT-3 models and organization keys.
- Easy-to-use event-based system for handling GPT-3 responses.
- Robust error handling for connection and request failures.
- Support for custom user-defined structs to JSON serialization and deserialization.
- Convenience methods for converting between JSON strings and user-defined structs.
- Debug logging and settings for enabling or disabling log output.

With the GameMindForge plugin, game developers can take their storytelling and character interaction to the next level, delivering immersive gameplay experiences that keep players engaged and coming back for more.

Here are just five example creative uses of the "UGameMindForge" plugin to enhance various aspects of a game:

1. NPC Conversations - Use the ChatGPT model to generate realistic conversations between NPCs and the player. This can add depth to the game world by allowing NPCs to have unique personalities, preferences, and goals.
2. Quest Design - Use the plugin to generate quest objectives, hints, and dialogue to make them feel more immersive and unique. With the ability to generate text based on user input, the plugin could also allow players to create their own quests or storylines.
3. Procedural Content - Use the plugin to generate random descriptions of items, enemies, and environments to make each playthrough feel fresh and unique. This can add replayability to the game and keep players engaged.
4. Puzzle Design - Use the plugin to generate hints, clues, and solutions to puzzles in the game. This can make puzzles more challenging and engaging, as well as give players a sense of accomplishment when they solve them.
5. Dynamic Narratives - Use the plugin to dynamically generate narratives based on the player's choices and actions in the game. This can create a personalized storytelling experience for each player, as well as create a sense of agency and impact on the game world.

Important Note Regarding Authentication

Please note that the GameMindForge plugin relies on the OpenAI API to generate responses to user input. Before using this plugin, you will need to set up an account with OpenAI and generate an APIKey and OrgKey to authenticate your requests.

To get started, please visit the OpenAI website and sign up for an account. Once you've created an account, you can navigate to the API dashboard to generate an API key and find your organization’s key.

https://platform.openai.com/account/api-keys

Once you have your APIKey and organization key, you can enter them into the GameMindForge component's properties in the blueprint editor. These properties will be used to authenticate your requests and generate responses from OpenAI's GPT models.

Plugin Settings

It’s necessary to configure your plugin settings when you first enable the plugin.

Open the Project Settings by clicking Edit > Project Settings in the main menu.
Expand the Plugins section and click on the GameMindForge settings.
In the GameMindForge settings, you will see the following options:

- **API Key**: The API key is a unique identifier issued by OpenAI that allows you to use the GPT-3 API. To get your API key, go to the OpenAI website, log in to your account, and navigate to the API Keys section. Copy the key and paste it into this field.
- **ORG Key**: The ORG key is an identifier issued by OpenAI that allows you to use the GPT-3 API within a specific organization. To get your ORG key, go to the OpenAI website, log in to your account, and navigate to the Organizations section. Copy the key and paste it into this field.
- **Authentication URL**: This is the URL of the OpenAI API endpoint that the plugin will use to authenticate and make requests to GPT-3. The default value is "https://api.openai.com/v1/chat/completions".
- **GPT Model:** This is the name of the GPT-3 model that the plugin will use to generate text. The default value is "gpt-3.5-turbo".
- **Show Debug**: This option enables debug logging for the plugin. When enabled, additional logging will be displayed in the output log. The default value is false.

Note: The default values for the API key and ORG key fields are empty strings. You must obtain and enter your own API and ORG keys in order to use the plugin with the GPT-3 API.

Component Overview

When you add the **GameMindForge_Component** to an actor in the Blueprint editor, the component will appear as a section in the Details panel on the right-hand side of the screen. The properties that you can edit for the component are:

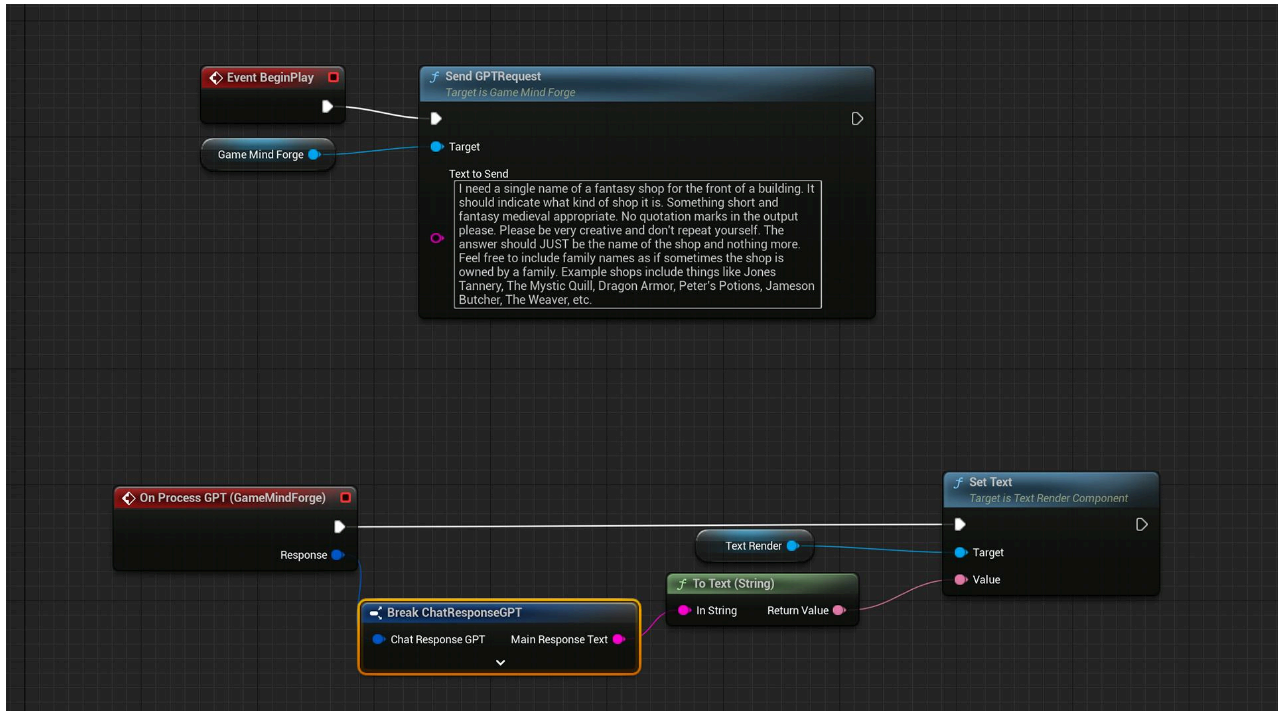
- ChatResponse (FChatResponseGPT): The response object returned by OpenAI after a GPT request is made. This property is read-only.
- MaxTokens (Integer): The maximum number of tokens to generate for a GPT request. Default value is **3000**.
- CreativityTemperature (Float): The creativity temperature to use for a GPT request. Default value is **0.8f**.

In the Blueprint editor, you can edit the values of these properties to customize the behavior of the **GameMindForge_Component**.

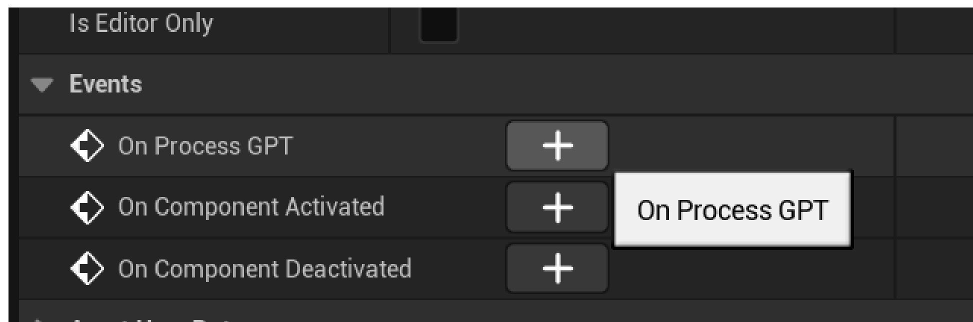
Usage Instructions - Simple Shop Title

In this section, we will demonstrate how to assign the GameMindForge Component to an actor in order to generate random shop names using OpenAI's GPT language model. The process will involve creating a new actor, adding the GameMindForge component to the actor, and setting up the component to generate greetings based on user input.

1. Create a new Blueprint Actor and name it "YeOldShoppe".
2. In the Components tab, add a Static Mesh Component and set its Static Mesh to a house or building mesh.
3. Add a Text Render Component and set its Location to (0,0,100) to place it above the store entrance. Set its Text to blank.
4. In the Event Graph, add a Begin Play event and create the following blueprint script:
    a. Call the "SendGPTRequest" function from the GameMindForge_Component, passing a prompt like the below screenshot.



b. Bind to the "OnProcessGPT" delegate event from the GameMindForge_Component and create a function to handle the response.



c. In the response handler function, parse the response and update the Text Render Component's Text property with the generated greeting.

5. Save and compile the Blueprint Actor.
6. Drag an instance of the blueprint into the level and press Play.
7. Wait for a moment and the Text Render Component should update with a new greeting generated by ChatGPT.

That's it! Now you have an NPC that greets players with a random message generated by ChatGPT.

Usage Instructions - More Complex Chat NPC

here are step-by-step instructions to create a more complex NPC that allows the user to chat with the NPC, pass input text, get responses, and keep talking:

1. Create a new actor called "ChatNPC" and add a Skeletal Mesh Component to it to add a visual representation of the NPC.
2. Create a new blueprint based on the ChatNPC actor and add a Text Render Component above the NPC's head to show the NPC's responses.
3. Add an instance of the GameMindForge_Component to the ChatNPC actor. This component will allow the NPC to generate responses using the ChatGPT model.
4. In the blueprint, add a new variable called "ChatHistory" of type "Array of Strings." This variable will keep track of the conversation history between the user and NPC.
5. Add a new function called "SendUserMessage" to the blueprint. This function will be called when the user enters a message to send to the NPC. The function should do the following:
    - Get the user's input text.
    - Add the user's message to the ChatHistory array.

    - Call the SendGPTRequest function of the GameMindForge_Component to generate a response from the ChatGPT model.
    - Bind a function to the OnProcessGPT delegate of the GameMindForge_Component to handle the response.
6. Add a new function called "HandleResponse" to the blueprint. This function will be called when the GameMindForge_Component generates a response from the ChatGPT model. The function should do the following:
    - Get the response text from the ChatResponse object.
    - Add the response text to the ChatHistory array.
    - Update the Text Render Component to show the NPC's response.
7. In the blueprint, add a new event called "BeginPlay" and do the following:
    - Set the Text Render Component to show a greeting message from the NPC.
    - Call the SendUserMessage function to start the conversation with the user.
8. Add a new function called "GetChatHistory" to the blueprint. This function will return the conversation history between the user and NPC stored in the ChatHistory array.
9. Add a new function called "ResetChatHistory" to the blueprint. This function will clear the ChatHistory array and reset the conversation with the NPC.

With these steps, you should be able to create a more complex NPC that allows the user to chat with the NPC, pass input text, get responses, and keep talking.

Usage Instructions - C++ Interaction

If you'd like to interact with GameMindForge in your C++ code, the process is simple. Here's an example:

This code demonstrates how to add an instance of the GameMindForge_Component to an actor class and interact with it to send requests and handle responses. First, we add a private member variable to our actor class to hold a pointer to the GameMindForge_Component. We then add a function to create and attach an instance of the component to the actor. In our example, we create an instance in the constructor of our actor class.

Once we have the component instance, we can use it to send requests to the OpenAI GPT API by calling its SendGPTRequest function. We pass the text we want to send as a parameter, and the component handles the HTTP request and response.

To handle the response, we define a function that will be called when the response is received. This function is bound to the OnProcessGPT delegate of the GameMindForge_Component using the AddDynamic function. When a response is received, the delegate is broadcasted and our function is called with the response object as a parameter.

Overall, this code demonstrates a simple but powerful way to use the GameMindForge plugin in C++ code.

Here's the header file for the example code:

```
C/C++
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "GameMindForge_Component.h"
#include "MyActor.generated.h"

UCLASS()
class AMyActor : public AActor
{
	GENERATED_BODY()

public:
	AMyActor();

protected:
	virtual void BeginPlay() override;

private:
	UGameMindForge* GameMindForgeComponent;

	void CreateGameMindForgeComponent();
	void HandleGPTResponse(FChatResponseGPT response);

};
```

And here's the corresponding CPP code.

```
C/C++
#include "MyActor.h"
```

```
AMyActor::AMyActor()
{
	CreateGameMindForgeComponent();
}

void AMyActor::CreateGameMindForgeComponent()
{
	GameMindForgeComponent = NewObject<UGameMindForge>(this,
TEXT("GameMindForgeComponent"));
	GameMindForgeComponent->RegisterComponent();
	GameMindForgeComponent->OnProcessGPT.AddDynamic(this,
&AMyActor::HandleGPTResponse);
}

void AMyActor::BeginPlay()
{
	Super::BeginPlay();

	// Send a GPT request using the component
	GameMindForgeComponent->SendGPTRequest("Hello, World!");
}

void AMyActor::HandleGPTResponse(FChatResponseGPT response)
{
	UE_LOG(LogTemp, Log, TEXT("Received GPT response with main
response text: %s"), *response.MainResponseText);
}
```

Here’s example code that creates a static mesh actor with a basic cube mesh, and above it a text render component, with the text set to a generated medieval fantasy greeting by the GameMindForge plugin.

C/C++

```
// Include necessary headers

#include "GameMindForge_Component.h"
#include "Components/StaticMeshComponent.h"
#include "Components/TextRenderComponent.h"

// In the constructor of your actor class, create the cube mesh
and text render component

AMyActor::AMyActor()
{

    // Create a cube mesh and set it as the root component
    UStaticMeshComponent* CubeMesh =
CreateDefaultSubobject<UStaticMeshComponent>(TEXT("CubeMesh"));

    RootComponent = CubeMesh;

    static ConstructorHelpers::FObjectFinder<UStaticMesh>
CubeMeshAsset(TEXT("/Engine/BasicShapes/Cube"));

    if (CubeMeshAsset.Succeeded())
    {

        CubeMesh->SetStaticMesh(CubeMeshAsset.Object);

    }

    // Create a text render component and attach it above the
cube mesh
```

```
    UTextRenderComponent* TextRenderComponent =
CreateDefaultSubobject<UTextRenderComponent>(TEXT("TextRenderComp
onent"));

    TextRenderComponent->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepRelativeTransform);

    TextRenderComponent->SetRelativeLocation(FVector(0.f, 0.f,
50.f));

    TextRenderComponent->SetTextRenderColor(FColor::White);
    TextRenderComponent->SetHorizontalAlignment(EHTA_Center);
    TextRenderComponent->SetVerticalAlignment(EVRTA_TextCenter);

    // Use GameMindForge to generate a medieval fantasy greeting
and set it as the text on the text render component

    UGameMindForge* GameMindForge =
NewObject<UGameMindForge>(this, TEXT("GameMindForge"));
    GameMindForge->SendGPTRequest("Generate a medieval fantasy
greeting");
    GameMindForge->OnProcessGPT.AddDynamic(this,
&AMyActor::OnGPTResponseReceived);

}

// Define a function that will be called when the GameMindForge
response is received

void AMyActor::OnGPTResponseReceived(FChatResponseGPT Response)
{
```

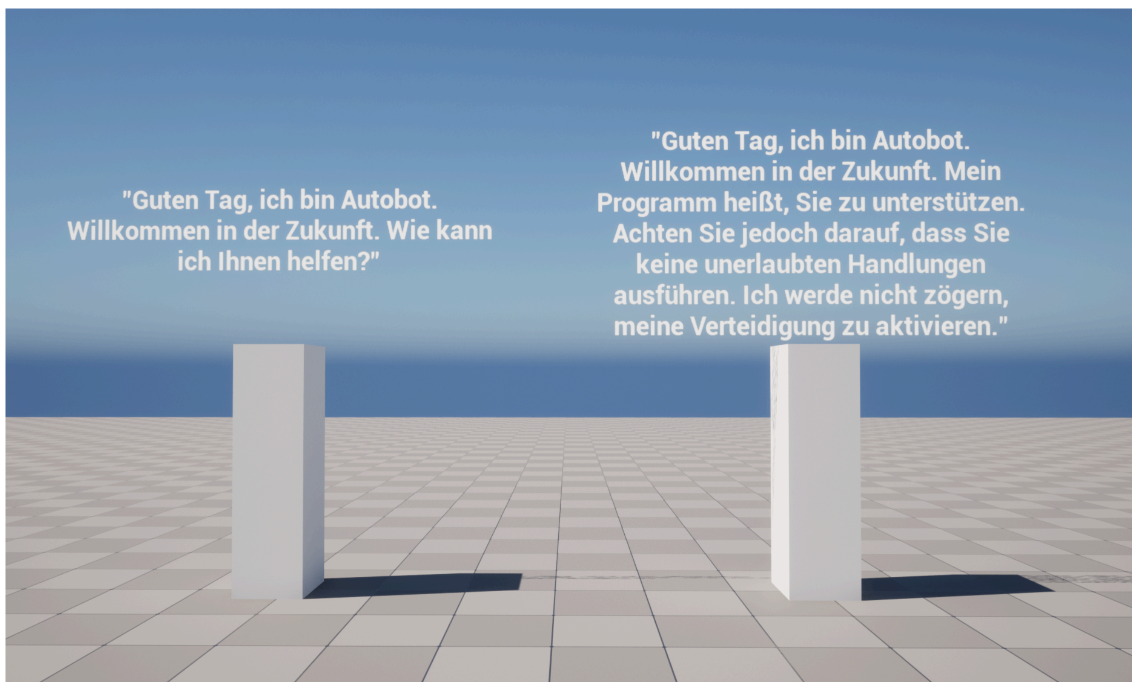
```
    // Get the text render component and set its text to the
generated greeting

    UTextRenderComponent* TextRenderComponent =
FindComponentByClass<UTextRenderComponent>();

    if (TextRenderComponent)
    {

        TextRenderComponent->SetText(Response.MainResponseText);

    }
}
```

Prompt Ideas and Variations:

Prompt:

```
Unset
You are an NPC called Autobot in a futuristic game. You're
somewhat like HAL 9000. Respond with a generic futuristic
sounding greeting. Don't respond with any variables. Always
introduce yourself as Autobot. Greet the player, but have a
sinister undertone like HAL 9000 indicating that the AI might be
sentient and defend itself. Write your response in German.
```

Output

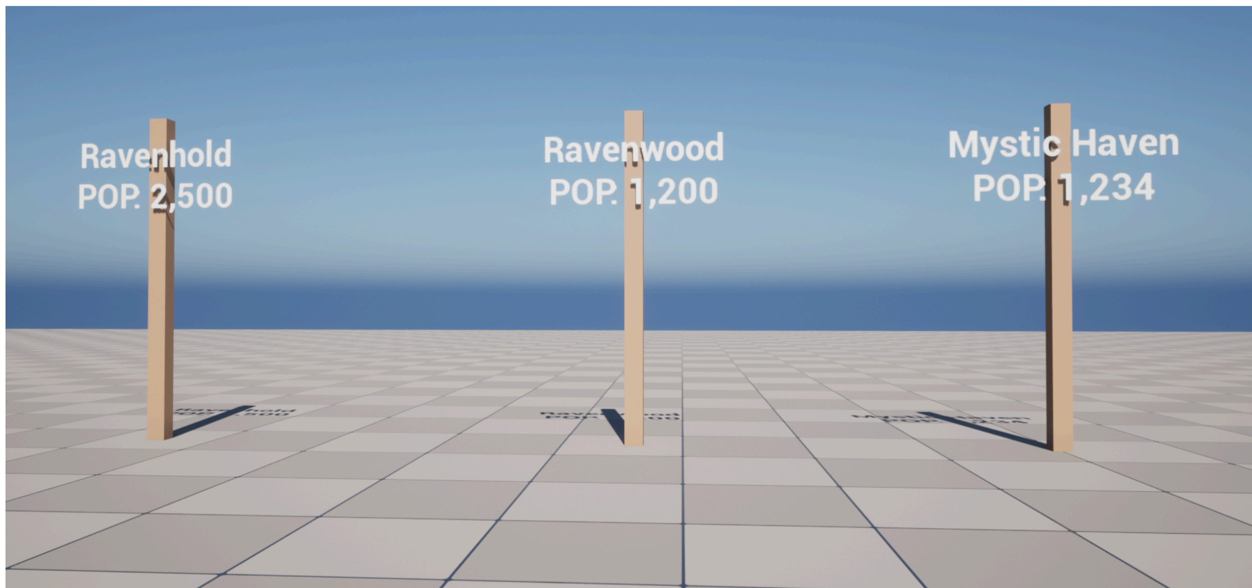


This is extremely powerful, as you can put any language as the target, and the NPC will instantly speak in that language.

Prompt

```
Unset
You are a road sign outside a village with the name of the
village on it. Make up a fantasy medieval village name and
population POP: on the next line. The output should only be the
village name, and then the POP on the next line.
```

Output



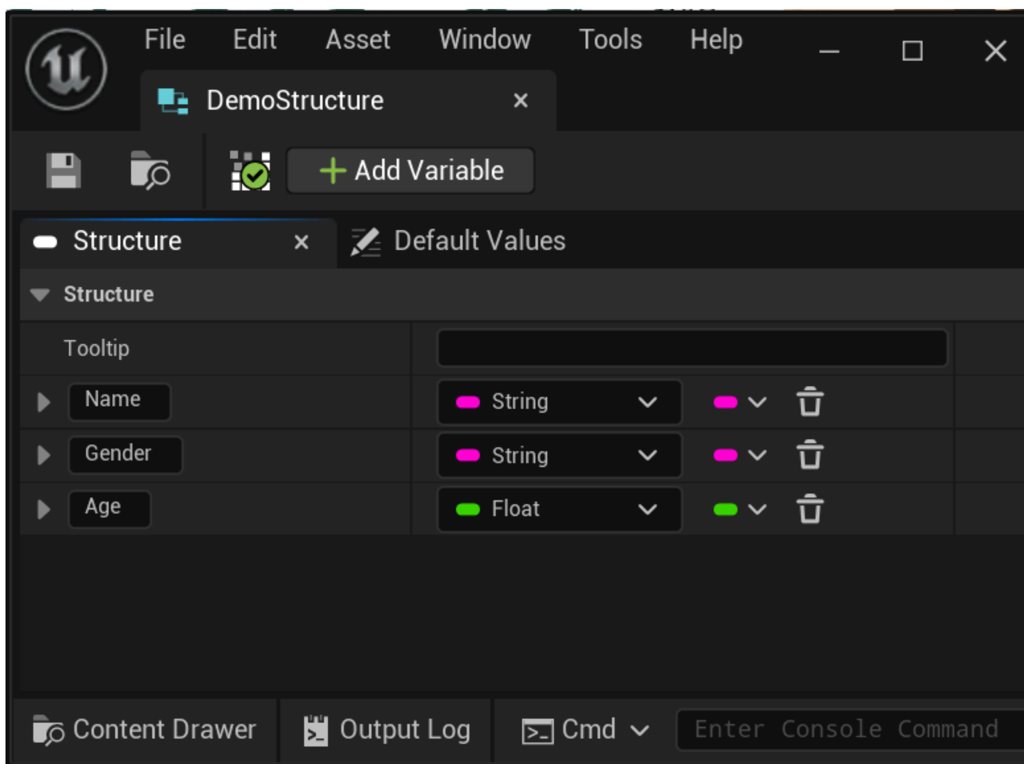
Auto Struct Generation and Population via JSON

With the ability to convert structures to JSON and back, the GameMindForge plugin opens up a world of possibilities for procedurally generated content. One of the most powerful aspects of the plugin is the ability to create JSON-formatted data on the fly, using ChatGPT to generate complex data structures. The plugin's ConvertStructToJSON and ConvertJsonToStruct functions are available as Blueprint nodes, allowing users to define and manipulate data structures entirely within Blueprint. This gives designers and developers a high level of flexibility and control over their game's content, making it possible to generate vast amounts of content automatically, while still ensuring that it conforms to the game's design specifications. This section of the documentation will explore the various ways that these functions can be used to create procedurally generated content, as well as provide guidance on how to use them effectively.

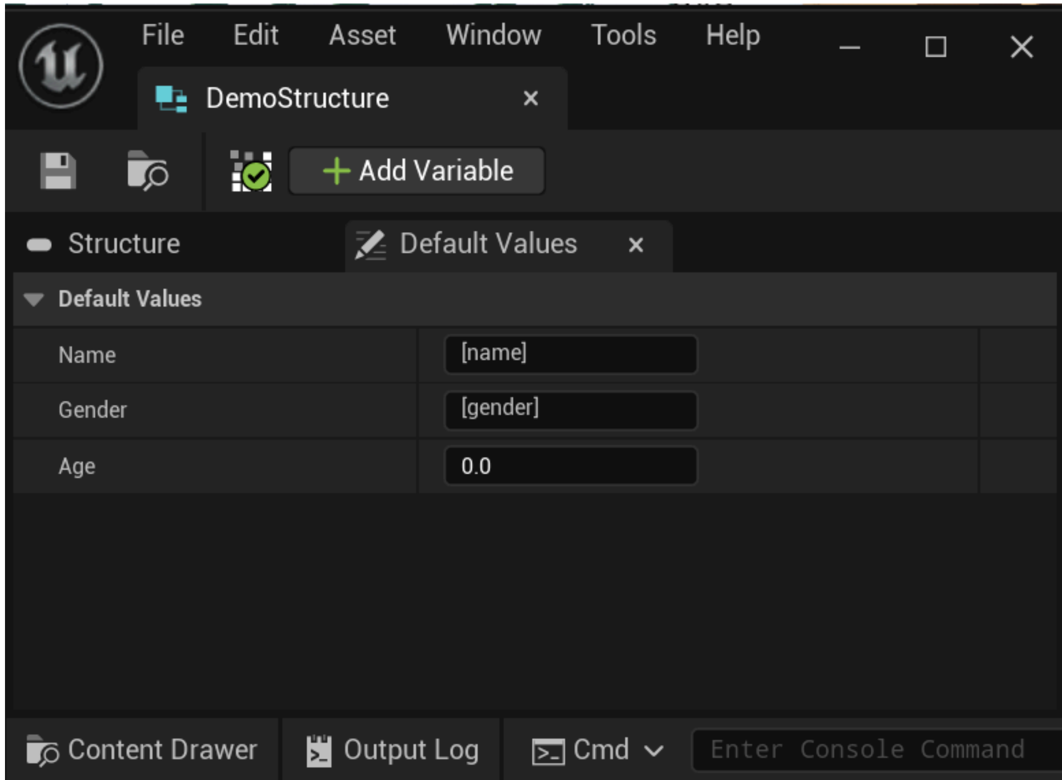
This means we can define a Blueprint struct, have the GameMindForge plugin teach ChatGPT how to format data, have ChatGPT generate data and return it in that format, and then use the plugin to convert the response from ChatGPT into a fully populated Blueprint structure variable, seamlessly.

Step 1

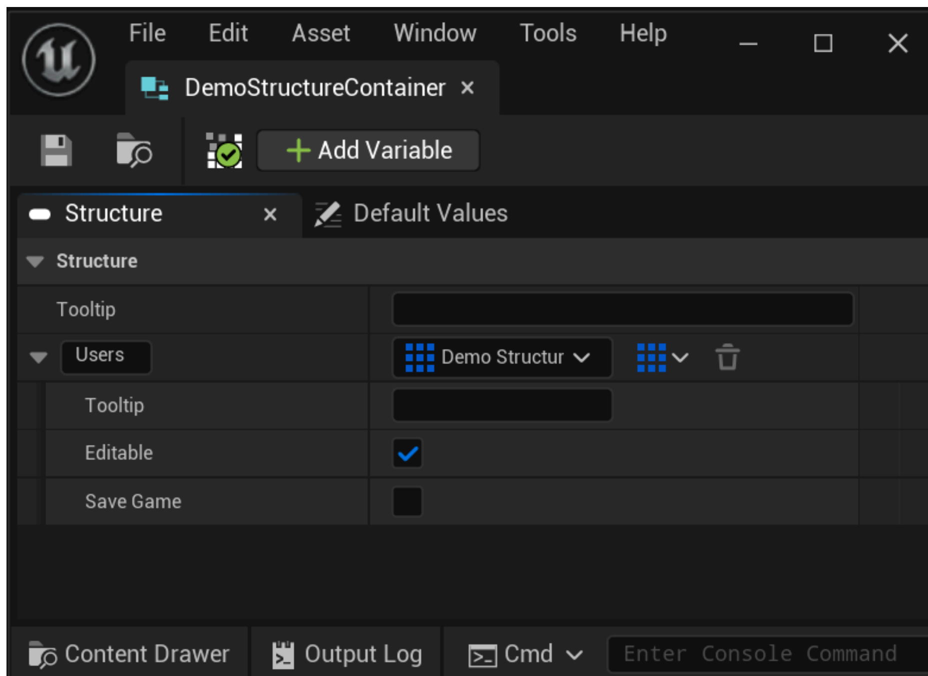
Create a Blueprint struct, or nested struct to define your data format. In this example, we'll be creating an array of users, each user with the attributes Name, Gender, and Age. In our example, first create a struct for the user data. We're calling this DemoStructure.



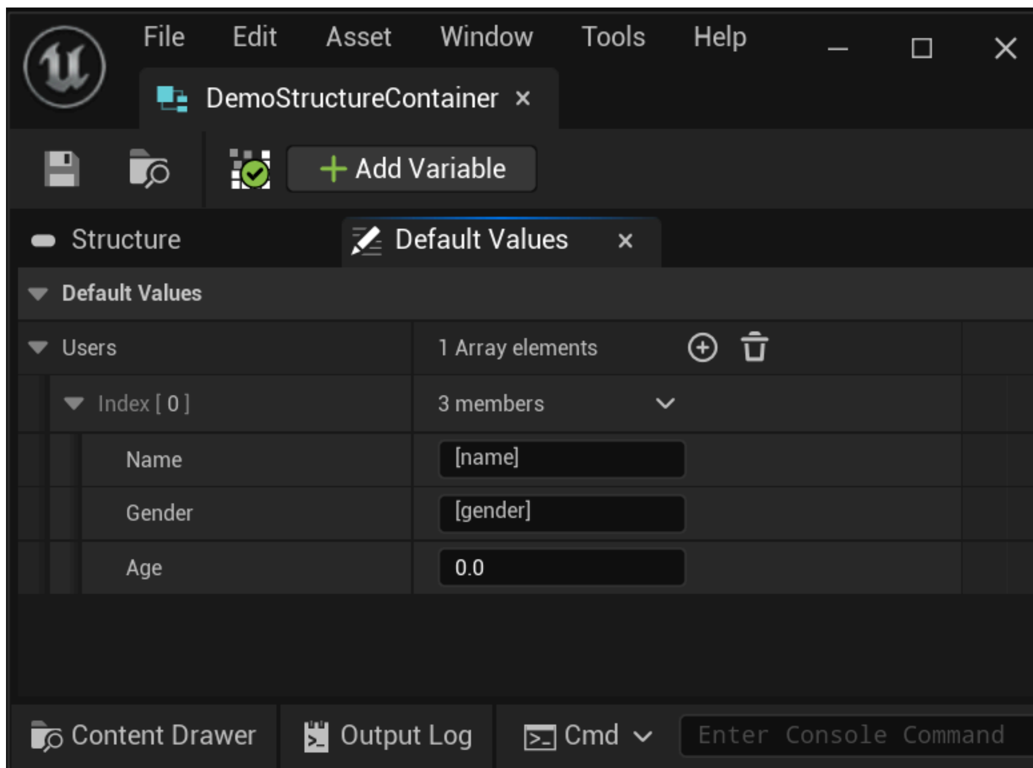
Note, it's very important to pre-populate the Default Values with values that match the struct value name, in square brackets. So, the Name field, which is an FString, is prepopulated with [name], like so:



Next, we create another structure called DemoStructureContainer, which in our case, is simply an array of DemoStructure.



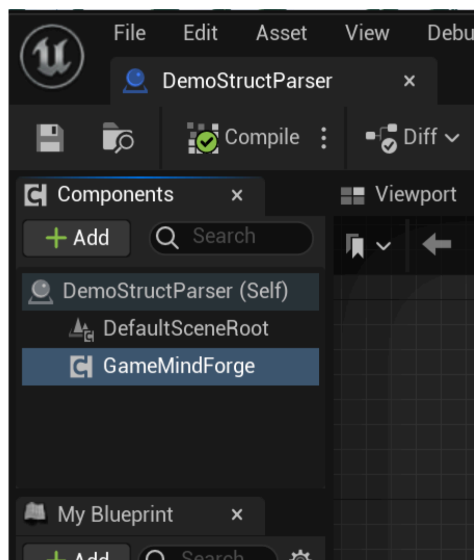
And again, populate it with at least one element, like so:



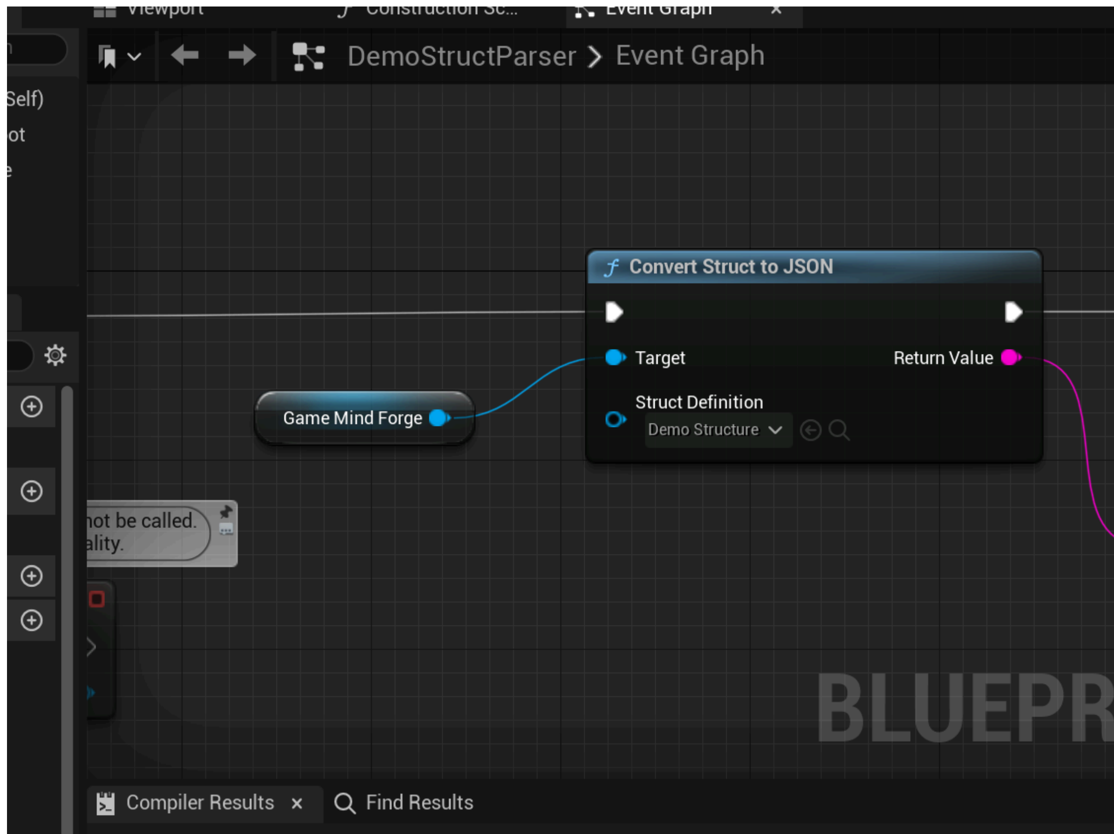
This is very important to be able to tell ChatGPT what our returned data structure should look like.

Step 2

Create a new blueprint, and attach the GameMindForge component to it.

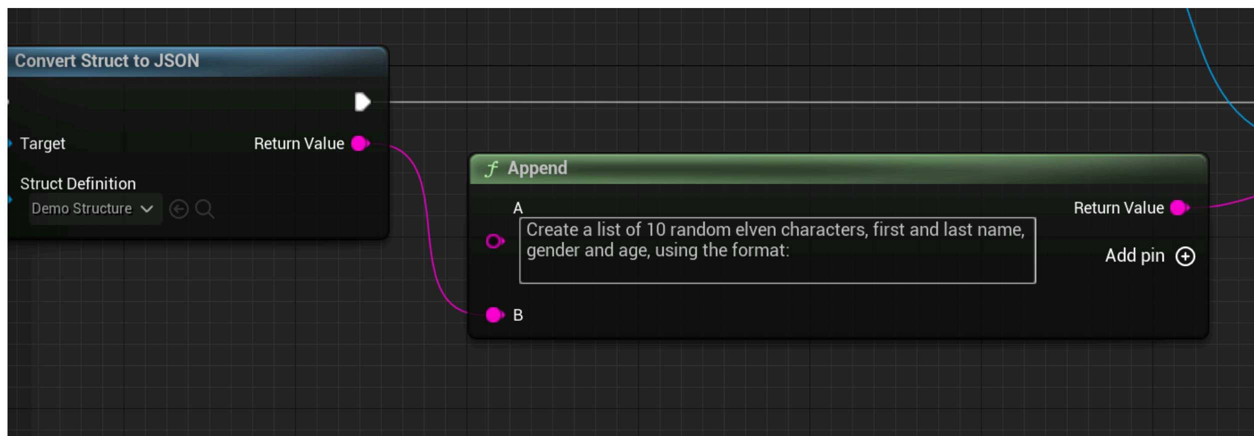


Drag an instance of the GameMindForge component into the graph, and off of it, pull the function ConvertStructToJSON. In the dropdown menu, choose the struct you created for the top level data, in this case, Demo Structure Container.



Step 3

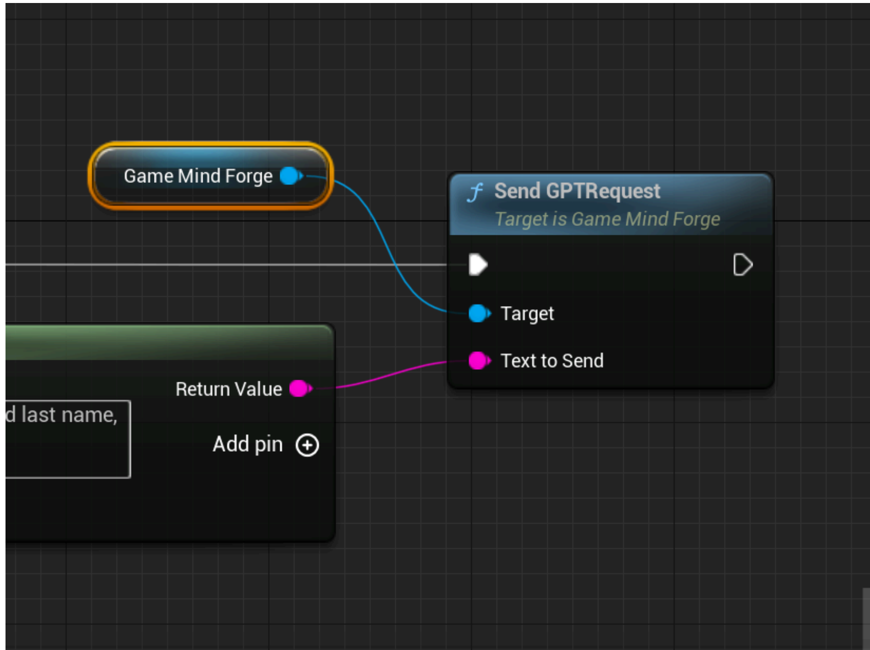
Drag from the string output, and into a new append string, create something like the following:



This will create an important string to be fed to ChatGPT. It will include the instructional prompt first, but then include a formatted and properly encoded representation of the Blueprint struct that will explain to ChatGPT how to format the response data.

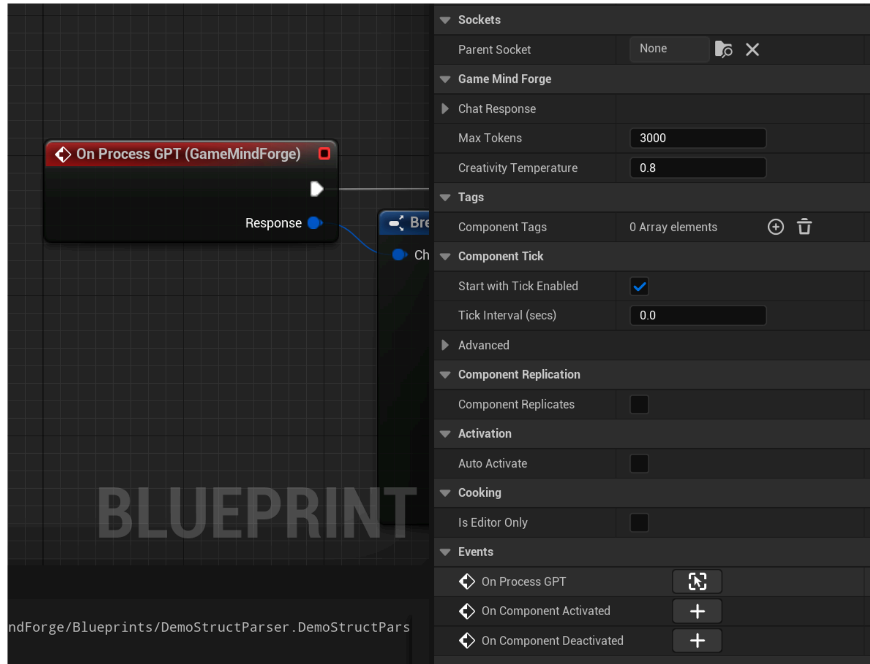
Step 4

Off of this Append node, drag the string into the SendGptRequest node of the GameMindForge component reference:



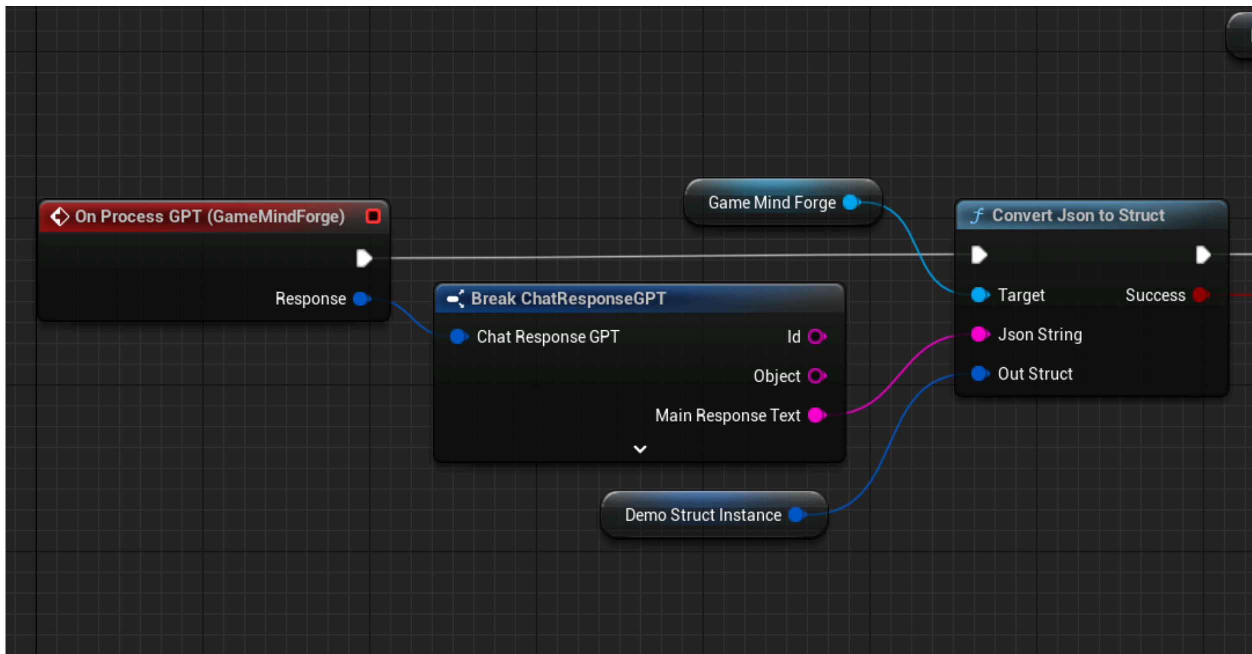
Step 5

Assign the OnProcessGPT event of the GameMindForge Component.



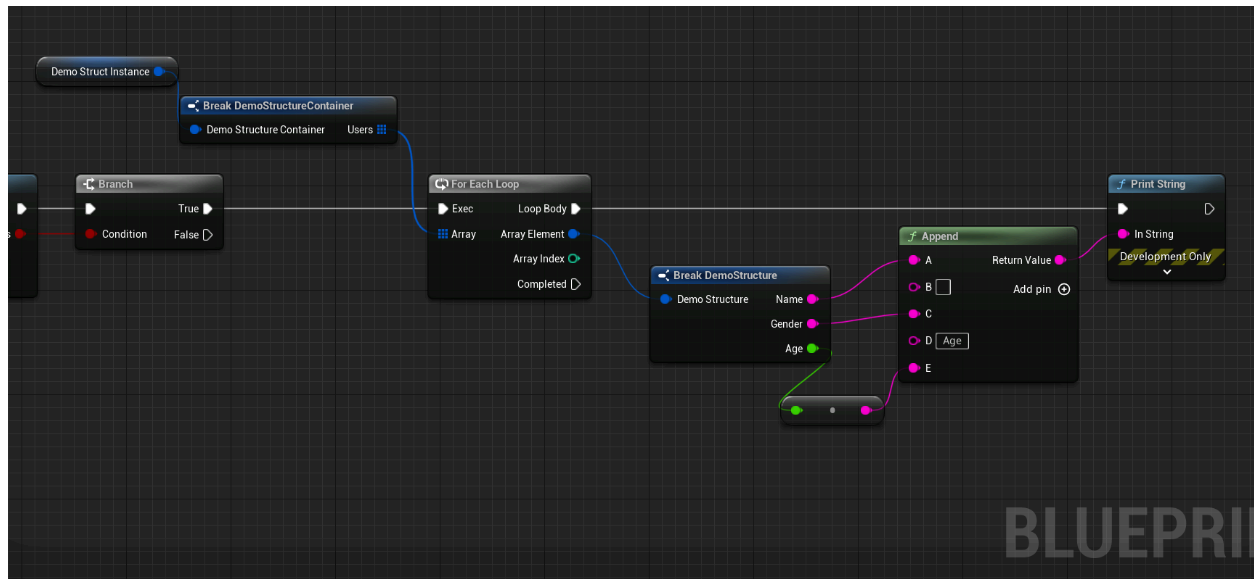
Step 6

From that node, drag the Response pin off and select Break pin struct and choose the Main Response Text value, and pass it into the ConvertJsonToStruct function of the GameMindForge Component. The function will automatically parse the result and find the JSON output. Into the node also drag an OutStruct, which will be the structure variable you want to populate.

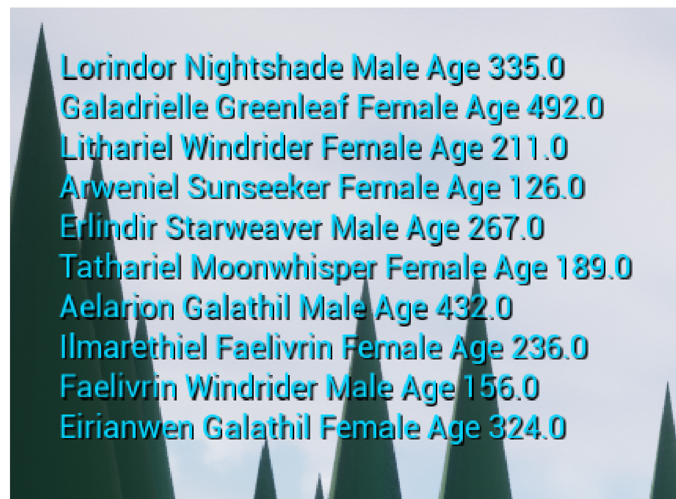


Step 7

Parse the results! The opportunities are endless. In this example, we simply print the result,



And this is the output we get:



Scratching the surface, here are some examples of things that you could use this powerful feature for to rapidly populate procedural content into your games:

- Define a character class with stats, abilities, and equipment, and get back a populated character instance.
- Define a level layout with spawn points, enemy types, and loot drops, and get back a JSON file that represents the level.
- Define a dialogue tree with characters, choices, and outcomes, and get back a JSON file that represents the dialogue sequence.

- Define a resource gathering system with resource types, gathering methods, and yield rates, and get back a JSON file that represents the resource gathering mechanics.
- Define a weapon class with stats, ammo types, and firing modes, and get back a populated weapon instance.
- Define a crafting system with recipe types, ingredient lists, and success rates, and get back a JSON file that represents the crafting mechanics.
- Define a mission structure with objectives, rewards, and failure conditions, and get back a JSON file that represents the mission structure.
- Define a status effect class with modifiers, durations, and triggers, and get back a populated status effect instance.
- Define a vehicle class with stats, handling characteristics, and weapon mounts, and get back a populated vehicle instance.
- Define a currency system with currency types, exchange rates, and item costs, and get back a JSON file that represents the in-game economy.

Advanced Prompt - Coding your own structs in C++

Here's an advanced prompt to make the response from ChatGPT into a more data-driven game element.

Unset

```
Create a quest for the player to complete.\n\nQuest Name: [quest
name]\nQuest Description: [quest
description]\n\nJSON_OUTPUT_START\n{\n  \"name\": \"[quest
name]\",\n  \"description\": \"[quest description]\",\n
\"required_items\": [\n    {\n      \"name\": \"[item name]\",\n
\"quantity\": [item quantity]\n    }\n  ],\n  \"location\": {\n
\"name\": \"[location name]\",\n    \"coordinates\": [\n
[latitude],\n      [longitude]\n    ]\n  },\n  \"rewards\": [\n
{\n      \"name\": \"[reward name]\",\n      \"quantity\":
[reward quantity]\n    }\n  ]\n}\nJSON_OUTPUT_END\n
```

This prompt is asking ChatGPT to create a quest (a task or objective) for a player to complete in a game. The quest will have a name, description, required items, a location, and rewards. The output format for the quest will be in JSON format.

Example output:

```
Unset
Quest Name: The Missing Heirloom

Quest Description: A wealthy merchant has lost a precious family
heirloom somewhere in the city. The merchant is offering a large
reward for the safe return of the item. You have been tasked with
finding the missing heirloom and returning it to the merchant.

JSON_OUTPUT_START
{
"name": "The Missing Heirloom",
"description": "A wealthy merchant has lost a precious family
heirloom somewhere in the city. The merchant is offering a large
reward for the safe return of the item. You have been tasked with
finding the missing heirloom and returning it to the merchant.",
"required_items": [
{
"name": "Map of the city",
"quantity": 1
}
],
"location": {
"name": "The city",
"coordinates": [
37.7749,
-122.4194
]
},
"rewards": [
{
"name": "Gold coins",
"quantity": 500
},
{
"name": "Experience points",
"quantity": 1000
```

```
}
]
}
JSON_OUTPUT_END
```

To use Unreal Engine C++ to parse the JSON response and get the data into Unreal Engine, you can follow these steps:

First, you will need to create a new C++ class in Unreal Engine. In Unreal Engine, go to File > New C++ Class and select "Actor" as the parent class.

Next, you will need to add a component to the actor to handle the parsing of the JSON data. In the header file for the actor, add the following code:

```
C/C++
#include "Json.h"
#include "JsonUtilities.h"

USTRUCT()
struct FQuestReward
{
    GENERATED_BODY()

    UPROPERTY()
    FString Name;

    UPROPERTY()
    int32 Quantity;
};

USTRUCT()
struct FQuestItem
{
    GENERATED_BODY()
```

```
    UPROPERTY()
    FString Name;

    UPROPERTY()
    int32 Quantity;
};

USTRUCT()
struct FQuestLocation
{
    GENERATED_BODY()

    UPROPERTY()
    FString Name;

    UPROPERTY()
    float Latitude;

    UPROPERTY()
    float Longitude;
};

USTRUCT()
struct FQuestData
{
    GENERATED_BODY()

    UPROPERTY()
    FString Name;

    UPROPERTY()
    FString Description;

    UPROPERTY()
    TArray<FQuestItem> RequiredItems;
```

```
    UPROPERTY()
    FQuestLocation Location;

    UPROPERTY()
    TArray<FQuestReward> Rewards;
};
```

This code defines the structures that will be used to store the quest data. Each structure corresponds to a part of the quest, such as the required items or the location.

In the actor's source file, add the following code to parse the JSON data:

```
C/C++
#include "Json.h"
#include "JsonUtilities.h"

void ParseQuestData(const FString& JsonData)
{
    FQuestData QuestData;

FJsonObjectConverter::JsonObjectStringToUStruct<FQuestData>(JsonD
ata, &QuestData, 0, 0);

    // Now you can use the quest data in Unreal Engine
    UE_LOG(LogTemp, Warning, TEXT("Quest Name: %s"),
*QuestData.Name);
}
```

This code defines a function to parse the JSON data and convert it into the FQuestData structure. The function takes an FString parameter called JsonData, which is the JSON data that was received from ChatGPT.

Finally, you can call the ParseQuestData function with the JSON data received from ChatGPT. This could be done in the actor's BeginPlay function or in response to a player triggering the quest.

```
C/C++
void AMyActor::BeginPlay()
{
    Super::BeginPlay();

    FString JsonString = "{ ... }"; // The JSON data received
from ChatGPT
    ParseQuestData(JsonString);
}
```

This code calls the ParseQuestData function with the JSON data received from ChatGPT. Replace "{ ... }" with the actual JSON data received.

That's it! Now you can use Unreal Engine C++ to parse the JSON response and get the data into Unreal Engine, from an AI-generated quest. Keep in mind, you'll need to parse out the string between the JSON_OUTPUT_START and JSON_OUTPUT_END, like so:

```
C/C++
int32 StartIndex = JsonString.Find(TEXT("JSON_OUTPUT_START"),
ESearchCase::IgnoreCase, ESearchDir::FromStart);
int32 EndIndex = JsonString.Find(TEXT("JSON_OUTPUT_END"),
ESearchCase::IgnoreCase, ESearchDir::FromEnd);
if (StartIndex != INDEX_NONE && EndIndex != INDEX_NONE &&
EndIndex > StartIndex)
{
    FString JsonData = JsonString.Mid(StartIndex + 17, EndIndex -
StartIndex - 17);
```

```
    FJsonObjectConverter::JsonObjectStringToUStruct(JsonData,
&QuestData, 0, 0);

    // Now the QuestData struct is populated with the JSON data
from the ChatGPT response
}
else
{
    UE_LOG(LogTemp, Error, TEXT("Failed to parse JSON data from
ChatGPT response."));
}
```

If you'd like to design a UStruct in blueprints, and have the Plugin parse that definition into a string you can pass to ChatGPT, use the function of the plugin called ConvertStructToJSON.

Final Thoughts

The integration of GPT with Unreal Engine marks a new era in the power and potential of AI in gaming and content creation. The ability to generate complex and nuanced responses in real-time, and to adapt to user input in a natural and human-like way, represents a leap forward in the sophistication and realism of game worlds and interactive experiences.

Imagine an NPC that can hold a fully-realized conversation with a player, responding to their questions and comments in a way that feels truly organic and immersive. Or a game that can seamlessly translate languages, enabling players from all over the world to communicate and interact in a shared virtual space.

With GPT integration, the possibilities for AI-powered gameplay and content generation are endless. Conversational agents, chatbots, and interactive storylines can all be designed to deliver dynamic and engaging experiences that respond to user input in real time.

Moreover, GPT-powered content can be formatted as JSON and integrated with other game systems to drive functionality and enhance user experience. For example, chat logs and conversation histories can be used to track player progress and personalize game experiences, while metadata and analytics can provide valuable insights into player behavior and preferences.

In short, the integration of GPT with Unreal Engine has the potential to transform the way we think about AI in gaming and content creation, opening up a world of possibilities for immersive, dynamic, and personalized experiences.

One example where GPT could be used to output a JSON formatted string containing a list of city names, populations, world coordinates, and road connections is in the creation of open-world games. In such games, the player is often tasked with exploring a vast, procedurally-generated landscape. To make this landscape feel more real and immersive, it is important to include details such as cities and towns, each with its own unique characteristics.

To accomplish this, the developer could use GPT to generate a JSON formatted string that contains a list of cities, their populations, their world coordinates, and the road connections between them. The developer could then use this information to create a virtual world that feels more organic and believable.

For example, GPT could be given a prompt such as: "Generate a list of 100 cities, along with their populations, coordinates, and road connections. The cities should be located within a 10,000 square mile area."

GPT could then generate a JSON formatted string that contains a list of 100 cities, each with its own name, population, latitude and longitude coordinates, and a list of the other cities it is connected to by road. This information could then be used to generate a virtual world that feels more immersive and believable.

In addition, the developer could use this same technique to generate other features of the game world, such as rivers, mountains, and forests, each with their own unique characteristics. By using GPT to generate this information, the developer can save time and effort in the content creation process, and focus on creating a more compelling and immersive game world.

Thank You!

Thank you for your support!

This plugin is a work in progress and will continue to be developed. Any suggestions are welcome.

Any feedback or suggestions are welcome at glenwrhodes@gmail.com