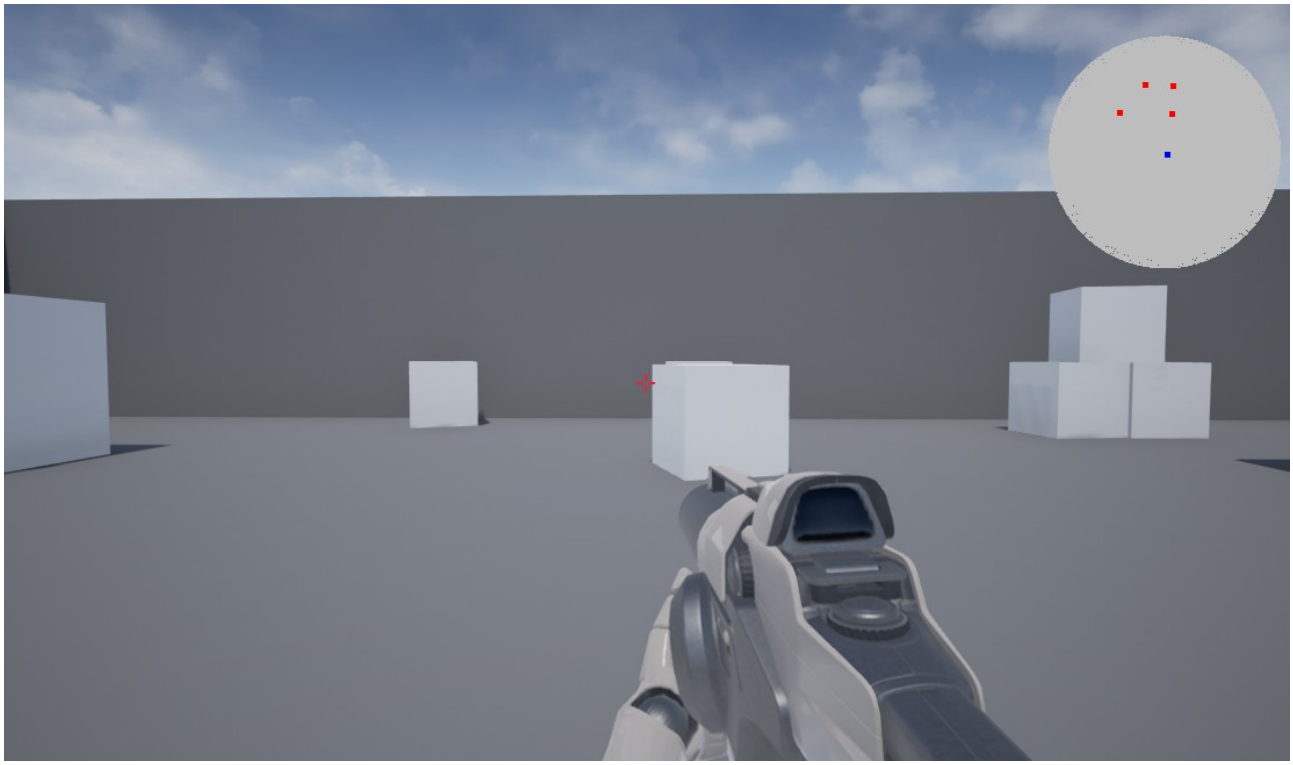
Introduction

(NB : J'espère que ce tutoriel vous sera utile et qu'il sera compréhensible par tous. Si vous avez des questions n'hésitez pas à m'envoyer un message sur Facebook « Antoine Gargasson » ou par mail antoine.gargasson@gmail.com)

(NB2 : Ce tuto est réalisé sur la version 4.11.2 de UE4)

Ce tutoriel est traduit de celui en anglais réalisé par Orfeas Eleftheriou.
Je vous encourage à vous rendre sur son site pour plus de tuto (en anglais) http://orfeasel.com/cpp-radar/

Dans ce tutoriel vous allez apprendre à créer un radar en C++ et créer des ennemies visibles sur ce radar. Vous pourrez par la suite personnaliser ce radar à votre convenance.



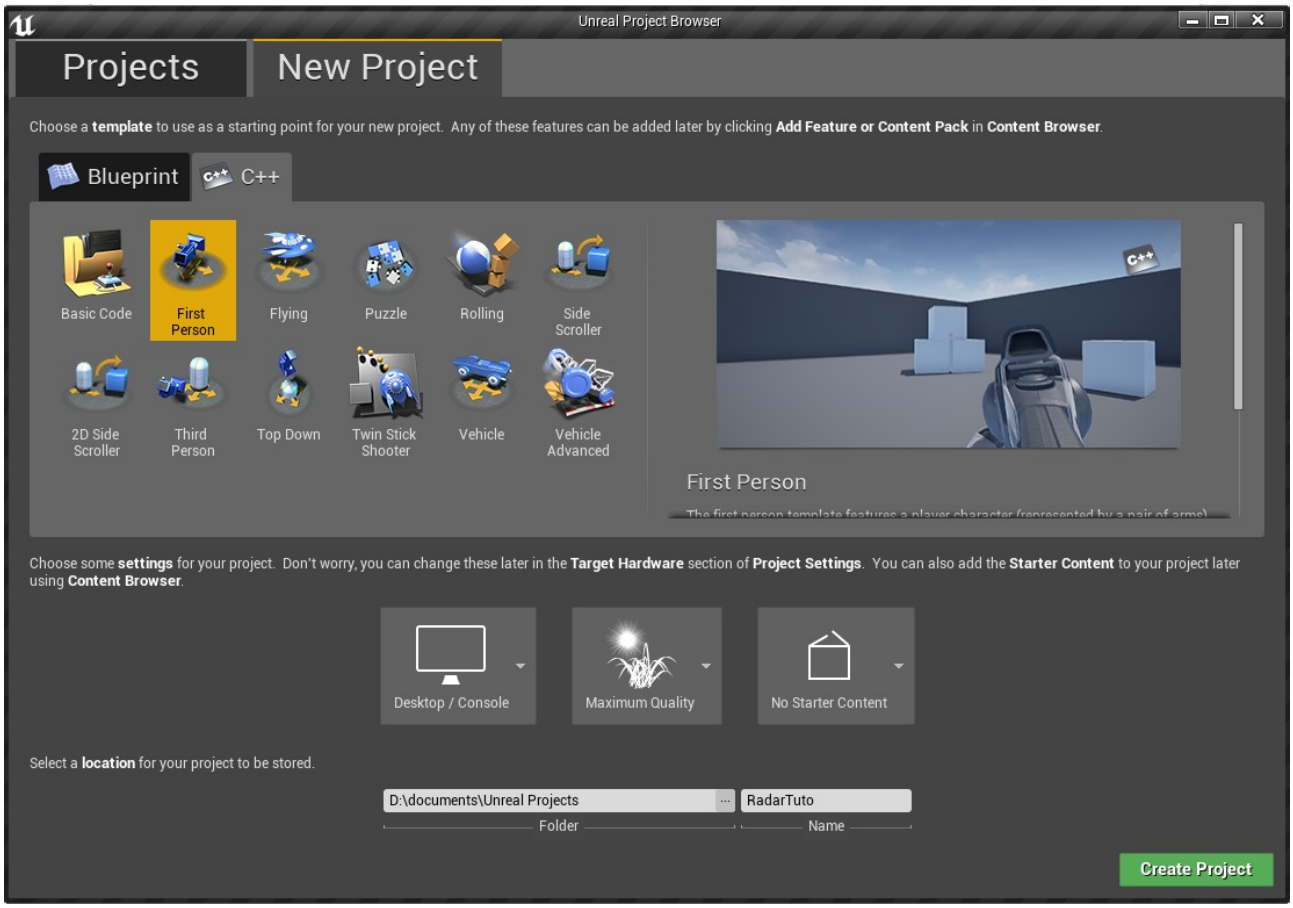
(en haut à droite vous pouvez voir le radar avec le point bleu au milieu représentant le joueur et chaque point rouge représentant les ennemies)

Pour un aperçu vidéo, rendez-vous à ce lien http://orfeasel.com/cpp-radar/

Préparation

Avant de vous lancer dans le code il vous faudra créer un projet 1ere personne en C++.

Ouvrez Unreal Engine 4.11 et sélectionnez le projet comme suit :



Unreal va vous ouvrir le projet ainsi que Visual studio 2015 !

Le HUD

Commençons par **dessiner le radar** !

Avant de dessiner quoi que ce soit, nous devons décider la position du radar sur l'écran du joueur.

Important à savoir : Tous les joueurs ne jouent pas avec la même résolution d'écran, nous devons donc dire à UE4 de dessiner le radar à une position relative à la résolution du joueur au lieu de le coder directement avec des valeurs fixes.

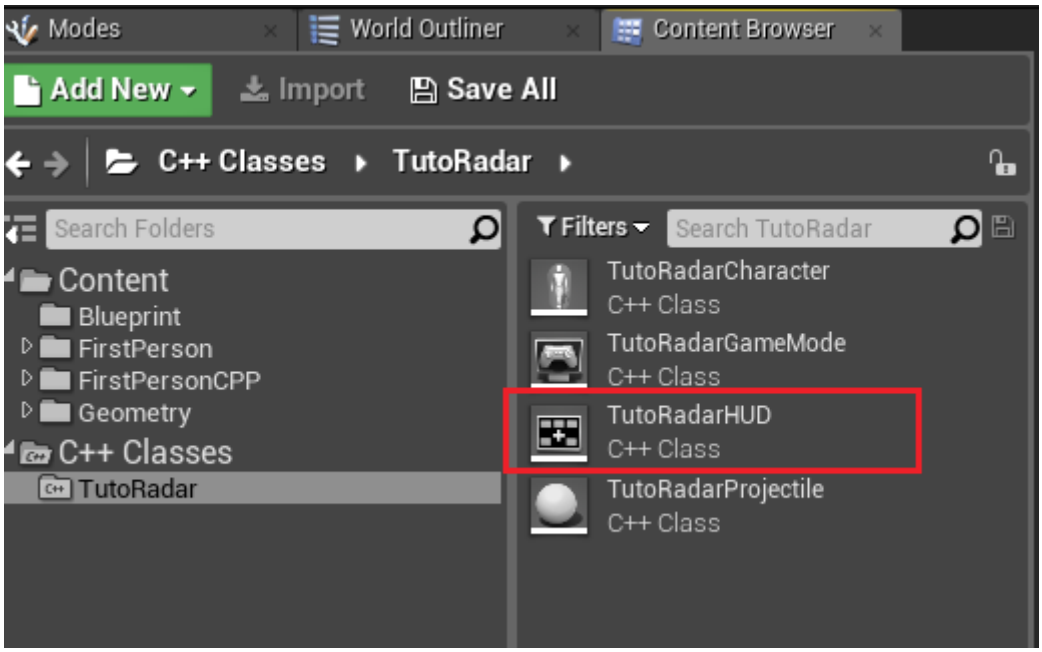
Fort heureusement, Epic a inclus dans la classe HUD (Head User Display, ou "affichage tête haute" en français) un canvas qui contient la dimension actuelle de l'écran du joueur (longueur et largeur). Afin de tirer parti de cette fonctionnalité, nous allons utiliser un vecteur 2D, que nous nommerons "RadarStartLocation", qui sera un multiplicateur afin de décider la position du radar en donnant des valeurs relatives à l'écran au lieu de mettre des valeurs fixes (ou magic number)

Considérons que notre résolution est de 1920 x 1080. La valeur 0 nous donnera le coin en haut à gauche de l'écran. Voici une image expliquant comment marche le système relatif :

Le X et le Y correspondent aux valeurs de notre multiplicateur (dans notre cas : RadarStartLocation). Si nous donnons une valeur de 0,9 pour X et de 0,2 pour Y, nous allons avoir notre radar placé en haut à droite de l'écran.
(Une fois le radar dessiné, il est conseillé de jouer un peu avec les valeurs afin d'avoir un positionnement optimal.)

Ces quelques explications données, il est temps de **commencer à coder notre radar** !

Commencez par ouvrir la classe HUD fournie avec le template de première personne en C++ que vous venez de créer.



Dans votre **HUD.h** rentrez ceci :

```
protected:

	/*la position de départ de notre radar*/
	UPROPERTY(EditAnywhere, Category = Radar)
	FVector2D RadarStartLocation = FVector2D(0.9f,0.2f);

	/*le rayon de notre radar*/
	UPROPERTY(EditAnywhere, Category = Radar)
	float RadarRadius = 100.f;

	/*le degré entre chaque étapes*/
	UPROPERTY(EditAnywhere, Category = Radar)
	float DegreeStep = 0.25f;

	/*la taille en pixel des acteurs à dessiner sur le radar*/
	UPROPERTY(EditAnywhere, Category = Radar)
	float DrawPixelSize = 5.f;
```

Puis rajoutez dans le .h les fonctions privées suivantes :

```
private:
	/*Renvoi le centre du radar sous forme de Vecteur 2D*/
	FVector2D GetRadarCenterPosition();

	/*Dessine le radar*/
	void DrawRadar();
```

Passons dans le **HUD.cpp** à présent. Nous allons écrire les deux fonctions que nous venons de rajouter en utilisant les variables rajoutées précédemment :

```
FVector2D AMinimapHUD::GetRadarCenterPosition()
{
	//Si le canvas est valide, renvoyer le centre en tant que vecteur 2D
	return (Canvas) ? FVector2D(Canvas->SizeX*RadarStartLocation.X, Canvas-
>SizeY*RadarStartLocation.Y) : FVector2D(0, 0);
}

void AMinimapHUD::DrawRadar()
{
	FVector2D RadarCenter = GetRadarCenterPosition();

	for (float i = 0; i < 360; i+=DegreeStep)
	{
	    //Nous voulons déssiner un cercle afin de représenter le radar
	    //Nous calculons donc le sinus et le cosinus pour presque chaque degrés
	    //C'est impossible de le calculer pour chaque degrés car ils sont infini
	    //Descendez la précision en degrés si vous avez besoin d'une plus grande précision
pour le cercle

	    //on multiplie nos coordonnées par la taille du radar
	    //afin de dessiner un cercle avec un rayon égal à celui que nous mettrons dans
l'éditeur
	      float fixedX = FMath::Cos(i) * RadarRadius;
	      float fixedY = FMath::Sin(i) * RadarRadius;

	      //Dessin actuel
	      DrawLine(RadarCenter.X, RadarCenter.Y, RadarCenter.X + fixedX, RadarCenter.Y +
fixedY, FLinearColor::Gray);
	}
}
```

Petites explications :

```
return (Canvas) ? FVector2D(Canvas->SizeX*RadarStartLocation.X, Canvas-
>SizeY*RadarStartLocation.Y) : FVector2D(0, 0);
```

Ceci est une "fonction Ternaire".
On l'écrit sous la forme suivante :

```
test qui renvoi un booléen ? Valeur si vrai : valeur si faux
```

Si un canvas existe on renvoi donc un vecteur qui est le centre de l'écran, sinon on renvoi la position 0. Si votre radar se dessine en haut à gauche, sachez donc que vous avez une erreur ici ! :D

Le drawLine prend les paramètres suivants :

1. Coordonnées X pour commencer la ligne
2. Coordonnées Y pour commencer la ligne
3. Coordonnées X pour finir la ligne
4. Coordonnées Y pour finir la ligne
5. La couleur de la ligne

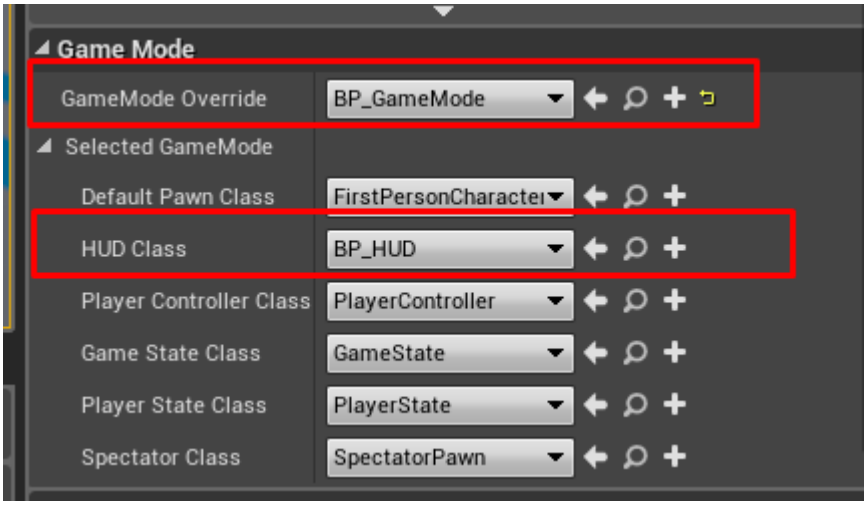
En considérant que vous avez compris et bien implémenté la logique ci-dessus, nous allons passer à la suite. Dans votre DrawHUD du .cpp, après ce qui a déjà été mis, rajoutez la ligne suivante :

```
DrawRadar();
```

Sauvegardez tout et compilez (Ctrl + Maj + b). Retournez enfin dans l'éditeur. Si tout est bon passez à la suite sinon relisez bien votre code ! :)

Avant de tester :

- Créez un blueprint de type GameMode basé sur le game mode par défaut donné dans le template.
- Créez un blueprint de type HUD basé sur votre classe C++ HUD.
- Assignez le game mode à votre projet et changez le hud de base par votre blueprint créé précédemment dans le "world settings" :



Vous pouvez également assigner la classe C++ directement pour le HUD mais vous ne pourrez pas changer les valeurs facilement si vous ne passez pas par un blueprint. Il est suggéré de suivre cette approche pour le début afin de ne pas avoir à compiler le code très souvent (ce qui peut être long).

Si vous lancez le jeu, vous devriez pouvoir voir votre radar maintenant ! :D

Félicitations ! Vous avez la base de votre radar !

Dans la suite du tuto, nous allons rajouter le dessin des ennemies.

Dessinons à présent le joueur !

Par convention, le joueur sera toujours dessiné au centre du radar.
Créez une fonction privée nommée DrawPlayerInRadar dans le HUD.h de type void.

Dans le HUD.h on a donc ceci :

```
void DrawPlayerInRadar();
```

Puis rajoutez dans HUD.cpp :

```
void AMinimapHUD::DrawPlayerInRadar()
{
	FVector2D RadarCenter = GetRadarCenterPosition();

	DrawRect(FLinearColor::Blue, RadarCenter.X, RadarCenter.Y, DrawPixelSize,
DrawPixelSize);
}
```

Ici on récupère le centre du radar et on dessine un rectangle bleu au centre du radar.

Pour finir avec le joueur, retournez dans la fonction DrawHUD dans le HUD.cpp et sous DrawRadar(), ajoutez "DrawPlayerInRadar();"

Vous avez à présent votre joueur qui est dessiné ! :D

Il nous reste à récupérer les ennemies et à les dessiner !

Récupérons les ennemies !

Dans ce cas, l'auteur a décidé d'utiliser un raycast pour plusieurs actors à côté du joueur et de référencer tous les actors qui contiennent le tag "Radar" dans un tableau afin de les dessiner sur le radar. Cependant, suivant votre jeu, votre cas peut varier ! Il est suggéré de suivre cette approche et qu'une fois le radar fonctionnel, d'implémenter votre logique.

Ceci dit, créez ces variables protected dans votre HUD.h :

```
/*hauteur de la sphère et son rayon pour le raycast*/
UPROPERTY(EditAnywhere, Category = Radar)
float SphereHeight = 200.f;

UPROPERTY(EditAnywhere, Category = Radar)
float SphereRadius = 2750.f;

/*Holds a reference to every actor we are currently drawing in our radar*/
TArray<AActor*> RadarActors
```

L'auteur recommande son tutoriel sur les raycast : http://orfeasel.com/tracing-multiple-objects/

Ensuite, créez la fonction privée suivante dans le HUD.h :

```
void PerformRadarRaycast() ;
```

Et voici l'implémentation dans le .cpp :

```
void AMinimapHUD::PerformRadarRaycast()
{
	APawn* Player = UGameplayStatics::GetPlayerPawn(GetWorld(), 0);

	if (Player)
	{
		TArray<FHitResult> HitResults;
		FVector EndLocation = Player->GetActorLocation();
		EndLocation.Z += SphereHeight;

		FCollisionShape CollisionShape;
		CollisionShape.ShapeType = ECollisionShape::Sphere;
		CollisionShape.SetSphere(SphereRadius);

		//Effectue l'appel à la fonction raycast en sphere

		GetWorld()->SweepMultiByChannel(HitResults, Player->GetActorLocation(),
EndLocation, FQuat::Identity, ECollisionChannel::ECC_WorldDynamic, CollisionShape);

		for (auto It : HitResults)
		{
			AActor* CurrentActor = It.GetActor();
			//Si l'acteur est taggé "Radar" on l'ajoute au tableau
			if (CurrentActor && CurrentActor->ActorHasTag("Radar"))
				RadarActors.Add(CurrentActor);
		}
	}
}
```

Dans cette fonction on initialise un tableau, on effectue un raycast en sphere autour du jouer et on récupère les acteurs concernés.

Une fois ceci fait, vous pouvez rajouter juste, après l'appel à la fonction DrawPlayerInRadar dans le DrawHUD(), l'appel à la fonction "PerformRadarRaycast() ;".

Vous pouvez à présent récupérer tous les acteurs mais pas les dessiner ! C'est la prochaine étape !

Dessinons enfin les ennemies !

Afin de dessiner les acteurs récupérés avec le raycast, nous allons créer deux fonctions :

- La première va convertir la position du monde des acteurs à la position locale, basé sur le joueur.
- La seconde va dessiner les acteurs dans le radar.

Déclarez les propriétés suivantes ainsi que la fonction dans le .h :

```
/*La distance du radar du joueur*/
UPROPERTY(EditAnywhere, Category = Radar)
float RadarDistanceScale = 25.f;

/*Convertie la position de l'acteur donné en local (basé sur le joueur)/*
FVector2D ConvertWorldLocationToLocal(AActor* ActorToPlace);

/*Dessine les acteurs raycastés sur le radar*/
void DrawRaycastedActors();
```

Dans le .cpp, vous pouvez rajouter le code suivant pour la fonction de conversion :

```
FVector2D AMinimapHUD::ConvertWorldLocationToLocal(AActor* ActorToPlace)
{
	APawn* Player = UGameplayStatics::GetPlayerPawn(GetWorld(), 0);

	if (Player && ActorToPlace)
	{
		//Convertie la position mondiale en locale, basée sur le joueur
		FVector ActorsLocal3dVector = Player-
>GetTransform().InverseTransformPosition(ActorToPlace->GetActorLocation());

		//Tourne le vecteur de 90 degrés dans le sens inverse des aiguilles d'une
montre pour avoir la bonne rotation sur le radar
		ActorsLocal3dVector = FRotator(0.f, -90.f,
0.f).RotateVector(ActorsLocal3dVector);

		//Applique la distance du radar
		ActorsLocal3dVector /= RadarDistanceScale;

		//Renvoi un vecteur 2D basé sur le vecteur 3D créé ci-dessus
		return FVector2D(ActorsLocal3dVector);
	}
	return FVector2D(0,0);
}
```

Pour la fonction de dessin des acteurs raycastés, dans le .cpp, écrivez le code suivant :

```
void AMinimapHUD::DrawRaycastedActors()
{
	FVector2D RadarCenter = GetRadarCenterPosition();

	for (auto It : RadarActors)
	{
		FVector2D convertedLocation = ConvertWorldLocationToLocal(It);

		//On veut clamper la position de nos acteurs afin d'être sûr qu'ils
		s'affichent dans le radar
		//On créer donc un vecteur temporaire afin d'accéder à la fonction
		GetClampedToMaxSize2d.
		//Cette fonction renvoie un vecteur clampé (si nécéssaire) afin de
		correspondre à notre longueur maximum
		FVector tempVector = FVector(convertedLocation.X, convertedLocation.Y, 0.f);

		//Soustrait la taille du pixel afin que le radar soit plus précis
		tempVector = tempVector.GetClampedToMaxSize2D(RadarRadius - DrawPixelSize);

		//Assigne le X et le Y convertis au vecteur que l'on désire afficher
		convertedLocation.X = tempVector.X;
		convertedLocation.Y = tempVector.Y;

		DrawRect(FLinearColor::Red, RadarCenter.X + convertedLocation.X,
RadarCenter.Y + convertedLocation.Y, DrawPixelSize, DrawPixelSize);
	}
}
```

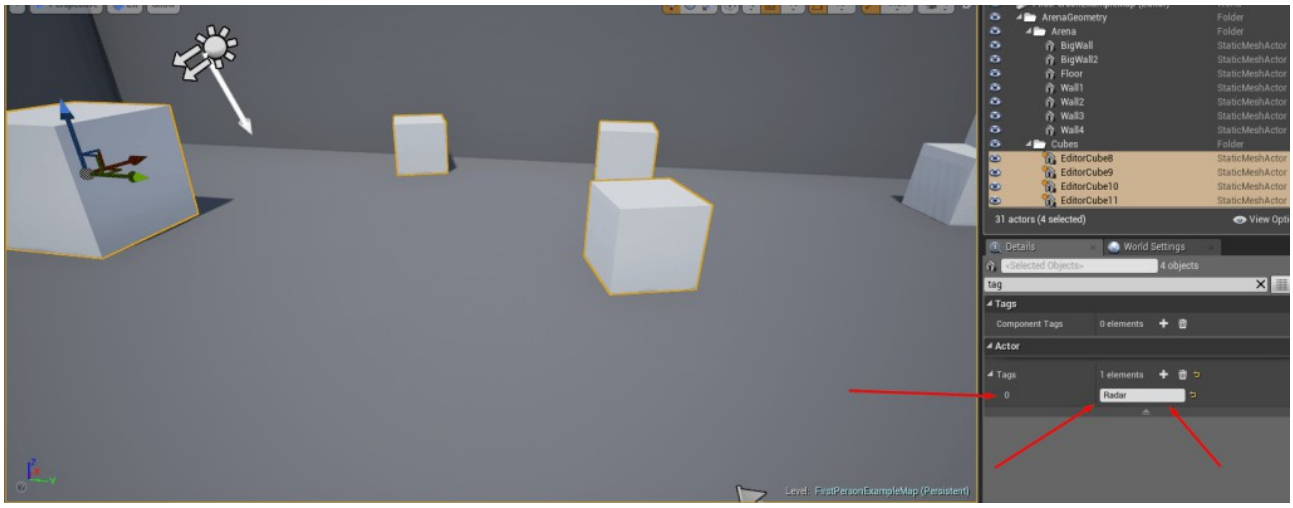
Une fois cette dernière fonction écrite, juste après le PerformRadarRaycast dans le DrawHUD, rajoutez le code suivant :

```
DrawRaycastedActors();

//Vide les acteurs du radars au cas ou le joueur se déplace hors de portée,
//en procédant comme ça, nous avons toujours un affichage correct du radar
RadarActors.Empty();
```

Pour finir voici l'implémentation complète du DrawHUD :

```
void AMinimapHUD::DrawHUD()
{
	//Default template code

	Super::DrawHUD();

	// Draw very simple crosshair

	// find center of the Canvas
	const FVector2D Center(Canvas->ClipX * 0.5f, Canvas->ClipY * 0.5f);

	// offset by half the texture's dimensions so that the center of the
texture aligns with the center of the Canvas
	const FVector2D CrosshairDrawPosition((Center.X), (Center.Y));

	// draw the crosshair
	FCanvasTileItem TileItem(CrosshairDrawPosition, CrosshairTex->Resource,
FLinearColor::White);
	TileItem.BlendMode = SE_BLEND_Translucent;
	Canvas->DrawItem(TileItem);


	//----------------Radar logic----------------

	DrawRadar();

	DrawPlayerInRadar();

	PerformRadarRaycast();

	DrawRaycastedActors();

	//Vide les acteurs du radars au cas ou le joueur se déplace hors de
	portée,
     //en procédant comme ça, nous avons toujours un affichage correct du radar
	RadarActors.Empty();
}
```

Sauvegardez et compilez votre code !
De retour dans l'éditeur, N'oubliez pas d'assigner le tag "Radar" aux acteurs !!



Vous voila avec un radar ! Vous pouvez l'agrémenter en mettant d'autres tags et fonctions !!
Vous pouvez par exemple rajouter d'autres objets, changer le tri, rajouter les alliés, etc.

Pour toute éventuelles questions, remarques, détails ou demandes, n'hésitez pas à prendre contact avec moi ! (Cf : début du tuto)