ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

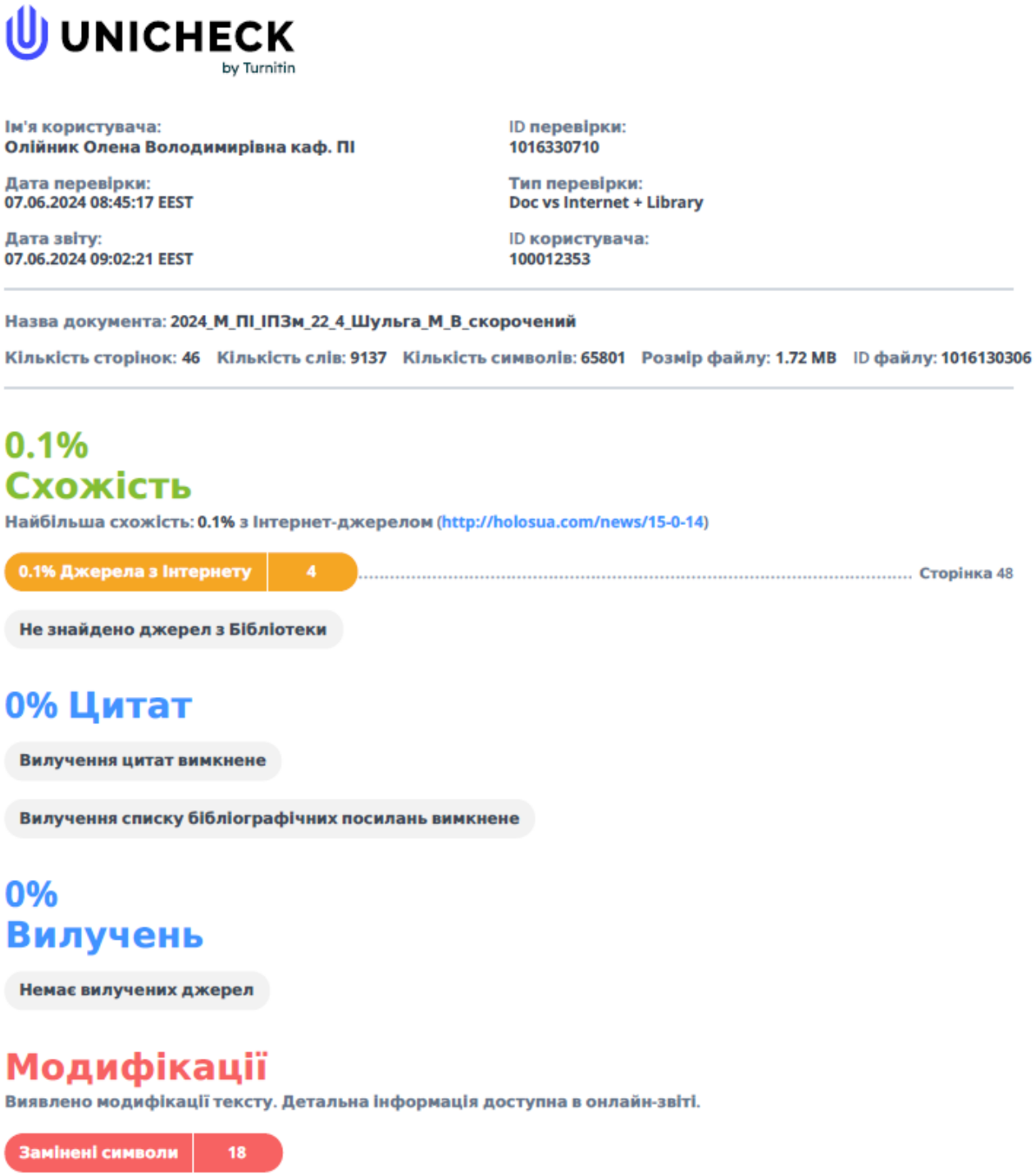

Рисунок А.1 - Результат перевірки кваліфікаційної роботи бакалавра на плагіат

ДОДАТОК Б

Програмний код

```
/*=============================================================================
====
      AnimationGraphSchema.cpp
=============================================================================
==*/

#include "AnimationGraphSchema.h"
#include "Animation/AnimationAsset.h"
#include "Animation/AnimBlueprint.h"
#include "Framework/MultiBox/MultiBoxBuilder.h"
#include "ToolMenus.h"
#include "ObjectEditorUtils.h"
#include "K2Node.h"
#include "EdGraphSchema_K2_Actions.h"
#include "Kismet2/BlueprintEditorUtils.h"
#include "Animation/AnimSequence.h"
#include "AnimStateNode.h"
#include "Animation/BlendSpace.h"
#include "Animation/AimOffsetBlendSpace.h"
#include "Animation/AimOffsetBlendSpace1D.h"
#include "Animation/AnimNodeBase.h"
#include "AnimGraphNode_Base.h"
#include "AnimGraphNode_AssetPlayerBase.h"
#include "AnimGraphNode_BlendSpacePlayer.h"
#include "AnimGraphNode_ComponentToLocalSpace.h"
#include "AnimGraphNode_LocalToComponentSpace.h"
#include "AnimGraphNode_Root.h"
#include "AnimGraphNode_RotationOffsetBlendSpace.h"
#include "AnimGraphNode_SequencePlayer.h"
#include "Animation/PoseAsset.h"
#include "AnimGraphNode_PoseBlendNode.h"
#include "AnimGraphNode_PoseByName.h"
#include "AnimGraphCommands.h"
#include "K2Node_Knot.h"
#include "ScopedTransaction.h"
#include "Animation/AnimMontage.h"
#include "AnimGraphNode_LinkedInputPose.h"
#include "AnimGraphNode_LinkedAnimLayer.h"
#include "AnimGraphNode_LinkedAnimGraphBase.h"
#include "AnimGraphNode_RigidBody.h"
#include "AnimationBlendSpaceSampleGraph.h"
#include "GraphEditorDragDropAction.h"
#include "AnimationEditorUtils.h"
#include "Settings/AnimBlueprintSettings.h"

#define LOCTEXT_NAMESPACE "AnimationGraphSchema"

namespace UE::Anim::BP::Editor
{
bool IsSkeletonCompatible(const UAnimBlueprint* AnimBlueprint, const
UAnimationAsset* Asset)
      {
            if (AnimBlueprint == nullptr)
```

```
            {
                  return false;    // No blueprint provided, cannot be
compatible
            }

            if (AnimBlueprint->bIsTemplate)
            {
                  return true;     // Templates are always compatible
            }

            if (AnimBlueprint->TargetSkeleton == nullptr)
            {
                  return false;    // No target skeleton provided, cannot be
compatible
            }

            return AnimBlueprint->TargetSkeleton-
>IsCompatibleForEditor(Asset->GetSkeleton());
      }
}

//////////////////////////////////////////////////
// FAnimationLayerDragDropAction
/** DragDropAction class for drag and dropping animation layers */
class ANIMGRAPH_API FAnimationLayerDragDropAction : public
FGraphSchemaActionDragDropAction
{
public:
      DRAG_DROP_OPERATOR_TYPE(FAnimationLayerDragDropAction,
FGraphSchemaActionDragDropAction)

      virtual FReply DroppedOnPanel(const TSharedRef< class SWidget >&
Panel, FVector2D ScreenPosition, FVector2D GraphPosition, UEdGraph& Graph)
override;
      virtual FReply DroppedOnNode(FVector2D ScreenPosition, FVector2D
GraphPosition) override;
      virtual FReply DroppedOnPin(FVector2D ScreenPosition, FVector2D
GraphPosition) override;
      virtual FReply DroppedOnAction(TSharedRef<FEdGraphSchemaAction>
Action) override;
      virtual FReply DroppedOnCategory(FText Category) override;
      virtual void HoverTargetChanged() override;

protected:

      /** Constructor */
      FAnimationLayerDragDropAction();

      static TSharedRef<FAnimationLayerDragDropAction>
New(TSharedPtr<FEdGraphSchemaAction> InAction, FName InFuncName,
UAnimBlueprint* InRigBlueprint, UAnimationGraph* InRigGraph);

      UAnimBlueprint* SourceAnimBlueprint;
      UAnimationGraph* SourceAnimLayerGraph;
      FName SourceFuncName;

      friend class UAnimationGraphSchema;
};
```

```
FAnimationLayerDragDropAction::FAnimationLayerDragDropAction()
    : FGraphSchemaActionDragDropAction()
    , SourceAnimBlueprint(nullptr)
    , SourceAnimLayerGraph(nullptr)
    , SourceFuncName(NAME_None)
{
}

FReply FAnimationLayerDragDropAction::DroppedOnPanel(const TSharedRef<
class SWidget >& Panel, FVector2D ScreenPosition, FVector2D GraphPosition,
UEdGraph& Graph)
{
      if (UAnimationGraph* TargetRigGraph = Cast<UAnimationGraph>(&Graph))
      {
            if (UAnimBlueprint* TargetAnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForGraph(TargetRig
Graph)))
            {
                  FGraphNodeCreator<UAnimGraphNode_LinkedAnimLayer>
LinkedInputLayerNodeCreator(*TargetRigGraph);
                  UAnimGraphNode_LinkedAnimLayer* LinkedAnimLayerNode =
LinkedInputLayerNodeCreator.CreateNode();
                  const FName GraphName = TargetRigGraph->GetFName();
                  LinkedAnimLayerNode->SetupFromLayerId(SourceFuncName);
                  LinkedInputLayerNodeCreator.Finalize();
                  LinkedAnimLayerNode->NodePosX =
static_cast<int32>(GraphPosition.X);
                  LinkedAnimLayerNode->NodePosY =
static_cast<int32>(GraphPosition.Y);

                  UBlueprint* Blueprint =
FBlueprintEditorUtils::FindBlueprintForGraphChecked(&Graph);
                  // See if we need to recompile skeleton after adding this
node, or just mark dirty
                  if (LinkedAnimLayerNode-
>NodeCausesStructuralBlueprintChange())
                  {

      FBlueprintEditorUtils::MarkBlueprintAsStructurallyModified(Blueprint)
;
                  }
                  else
                  {

      FBlueprintEditorUtils::MarkBlueprintAsModified(Blueprint);
                  }
            }
      }
      return FReply::Unhandled();
}

FReply FAnimationLayerDragDropAction::DroppedOnNode(FVector2D
ScreenPosition, FVector2D GraphPosition)
{
      if (UEdGraphNode* TargetNode = GetHoveredNode())
      {
```

```
            if (UAnimGraphNode_LinkedAnimLayer* LinkedAnimLayer =
Cast<UAnimGraphNode_LinkedAnimLayer>(TargetNode))
            {
                  LinkedAnimLayer->Node.Layer = SourceFuncName;
                  return FReply::Handled();
            }
      }
      return FReply::Unhandled();
}

FReply FAnimationLayerDragDropAction::DroppedOnPin(FVector2D
ScreenPosition, FVector2D GraphPosition)
{
      return FReply::Unhandled();
}

FReply
FAnimationLayerDragDropAction::DroppedOnAction(TSharedRef<FEdGraphSchemaAct
ion> Action)
{
      return FReply::Unhandled();
}

FReply FAnimationLayerDragDropAction::DroppedOnCategory(FText Category)
{
      return FReply::Unhandled();
}

void FAnimationLayerDragDropAction::HoverTargetChanged()
{
      FGraphSchemaActionDragDropAction::HoverTargetChanged();
      bDropTargetValid = true;
}


TSharedRef<FAnimationLayerDragDropAction>
FAnimationLayerDragDropAction::New(TSharedPtr<FEdGraphSchemaAction>
InAction, FName InFuncName, UAnimBlueprint* InAnimBlueprint,
UAnimationGraph* InAnimationLayerGraph)
{
      TSharedRef<FAnimationLayerDragDropAction> Action = MakeShareable(new
FAnimationLayerDragDropAction);
      Action->SourceAction = InAction;
      Action->SourceAnimBlueprint = InAnimBlueprint;
      Action->SourceAnimLayerGraph = InAnimationLayerGraph;
      Action->SourceFuncName = InFuncName;
      Action->Construct();
      return Action;
}

//////////////////////////////////////////////////
// UAnimationGraphSchema

UAnimationGraphSchema::UAnimationGraphSchema(const FObjectInitializer&
ObjectInitializer)
      : Super(ObjectInitializer)
{
      PN_SequenceName = TEXT("Sequence");
```

```
	NAME_NeverAsPin = TEXT("NeverAsPin");
	NAME_PinHiddenByDefault = TEXT("PinHiddenByDefault");
	NAME_PinShownByDefault = TEXT("PinShownByDefault");
	NAME_AlwaysAsPin = TEXT("AlwaysAsPin");
	NAME_OnEvaluate = TEXT("OnEvaluate");
	NAME_CustomizeProperty = TEXT("CustomizeProperty");
	DefaultEvaluationHandlerName = TEXT("EvaluateGraphExposedInputs");
}

FLinearColor UAnimationGraphSchema::GetPinTypeColor(const FEdGraphPinType&
PinType) const
{
	const bool bAdditive = PinType.PinSubCategory == TEXT("Additive");
	if (UAnimationGraphSchema::IsLocalSpacePosePin(PinType))
	{
		if (bAdditive)
		{
			return FLinearColor(0.12, 0.60, 0.10);
		}
		else
		{
			return FLinearColor::White;
		}
	}
	else if (UAnimationGraphSchema::IsComponentSpacePosePin(PinType))
	{
		//@TODO: Pick better colors
		if (bAdditive)
		{
			return FLinearColor(0.12, 0.60, 0.60);
		}
		else
		{
			return FLinearColor(0.20f, 0.50f, 1.00f);
		}
	}

	return Super::GetPinTypeColor(PinType);
}

EGraphType UAnimationGraphSchema::GetGraphType(const UEdGraph* TestEdGraph)
const
{
	return GT_Animation;
}

void UAnimationGraphSchema::CreateDefaultNodesForGraph(UEdGraph& Graph)
const
{
	// Create the result node
	FGraphNodeCreator<UAnimGraphNode_Root> NodeCreator(Graph);
	UAnimGraphNode_Root* ResultSinkNode = NodeCreator.CreateNode();
	NodeCreator.Finalize();
	SetNodeMetaData(ResultSinkNode, FNodeMetadata::DefaultGraphNode);
}
```

```
void UAnimationGraphSchema::HandleGraphBeingDeleted(UEdGraph&
GraphBeingRemoved) const
{
	if (UBlueprint* Blueprint =
FBlueprintEditorUtils::FindBlueprintForGraph(&GraphBeingRemoved))
	{
		// Look for state nodes that reference this graph
		TArray<UAnimStateNodeBase*> StateNodes;

	FBlueprintEditorUtils::GetAllNodesOfClassEx<UAnimStateNode>(Blueprint
, StateNodes);

		TSet<UAnimStateNodeBase*> NodesToDelete;
		for (int32 i = 0; i < StateNodes.Num(); ++i)
		{
			UAnimStateNodeBase* StateNode = StateNodes[i];
			if (StateNode->GetBoundGraph() == &GraphBeingRemoved)
			{
				NodesToDelete.Add(StateNode);
			}
		}

		// Delete the node that owns us
		ensure(NodesToDelete.Num() <= 1);
		for (TSet<UAnimStateNodeBase*>::TIterator It(NodesToDelete); It;
++It)
		{
			UAnimStateNodeBase* NodeToDelete = *It;

			FBlueprintEditorUtils::RemoveNode(Blueprint, NodeToDelete,
true);

			// Prevent re-entrancy here
			NodeToDelete->ClearBoundGraph();
		}

		// Remove pose watches from nodes in this graph
		if (UAnimBlueprint* AnimBlueprint =
Cast<UAnimBlueprint>(Blueprint))
		{

	AnimationEditorUtils::RemovePoseWatchesFromGraph(AnimBlueprint,
&GraphBeingRemoved);
		}
	}
}

bool UAnimationGraphSchema::IsPosePin(const FEdGraphPinType& PinType)
{
	return IsLocalSpacePosePin(PinType) ||
IsComponentSpacePosePin(PinType);
}

bool UAnimationGraphSchema::IsLocalSpacePosePin(const FEdGraphPinType&
PinType)
{
	UScriptStruct* PoseLinkStruct = FPoseLink::StaticStruct();
```

```
	return (PinType.PinCategory == UAnimationGraphSchema::PC_Struct) &&
(PinType.PinSubCategoryObject == PoseLinkStruct);
}

bool UAnimationGraphSchema::IsComponentSpacePosePin(const FEdGraphPinType&
PinType)
{
	UScriptStruct* ComponentSpacePoseLinkStruct =
FComponentSpacePoseLink::StaticStruct();
	return (PinType.PinCategory == UAnimationGraphSchema::PC_Struct) &&
(PinType.PinSubCategoryObject == ComponentSpacePoseLinkStruct);
}

FEdGraphPinType UAnimationGraphSchema::MakeLocalSpacePosePin()
{
	FEdGraphPinType PinType;
	PinType.ResetToDefaults();

	PinType.PinCategory = UAnimationGraphSchema::PC_Struct;
	PinType.PinSubCategoryObject = FPoseLink::StaticStruct();

	return PinType;
}

FEdGraphPinType UAnimationGraphSchema::MakeComponentSpacePosePin()
{
	FEdGraphPinType PinType;
	PinType.ResetToDefaults();

	PinType.PinCategory = UAnimationGraphSchema::PC_Struct;
	PinType.PinSubCategoryObject =
FComponentSpacePoseLink::StaticStruct();

	return PinType;
}

bool UAnimationGraphSchema::TryCreateConnection(UEdGraphPin* A,
UEdGraphPin* B) const
{
	UEdGraphPin* OutputPin = nullptr;
	UEdGraphPin* InputPin = nullptr;

	if(A->Direction == EEdGraphPinDirection::EGPD_Output)
	{
		OutputPin = A;
		InputPin = B;
	}
	else
	{
		OutputPin = B;
		InputPin = A;
	}
	check(OutputPin && InputPin);

	UK2Node_Knot* OutputKnotNode = Cast<UK2Node_Knot>(OutputPin-
>GetOwningNode());
	UK2Node_Knot* InputKnotNode = Cast<UK2Node_Knot>(InputPin-
>GetOwningNode());
```

```
	bool bConnectionWithKnot = OutputKnotNode != nullptr || InputKnotNode
!= nullptr;

	if(bConnectionWithKnot)
	{
		// Double check this is our "exec"-like line
		bool bOutputIsPose = IsPosePin(OutputPin->PinType);
		bool bInputIsPose = IsPosePin(InputPin->PinType);
		bool bHavePosePin = bOutputIsPose || bInputIsPose;
		bool bHaveWildPin = InputPin->PinType.PinCategory == PC_Wildcard
|| OutputPin->PinType.PinCategory == PC_Wildcard;

		if((bOutputIsPose && bInputIsPose) || (bHavePosePin &&
bHaveWildPin))
		{
			// Ok this is a valid exec-like line, we need to kill any
connections already on the output pin
			OutputPin->BreakAllPinLinks();
		}
	}

	if(Super::TryCreateConnection(A, B))
	{
		// Connection made - remove any bindings on the input pin
		if(UAnimGraphNode_Base* AnimGraphNode =
Cast<UAnimGraphNode_Base>(InputPin->GetOwningNode()))
		{
			// Compare FName without number to make sure we catch
array properties that are split into multiple pins
			FName ComparisonName = InputPin->GetFName();
			ComparisonName.SetNumber(0);
			AnimGraphNode->RemoveBindings(ComparisonName);
		}

		return true;
	}

	return false;
}

const FPinConnectionResponse
UAnimationGraphSchema::DetermineConnectionResponseOfCompatibleTypedPins(con
st UEdGraphPin* PinA, const UEdGraphPin* PinB, const UEdGraphPin* InputPin,
const UEdGraphPin* OutputPin) const
{
	// Enforce a tree hierarchy; where poses can only have one output
(parent) connection
	if (IsPosePin(OutputPin->PinType) && IsPosePin(InputPin->PinType))
	{
		if ((OutputPin->LinkedTo.Num() > 0) || (InputPin->LinkedTo.Num()
> 0))
		{
			const ECanCreateConnectionResponse ReplyBreakOutputs =
CONNECT_RESPONSE_BREAK_OTHERS_AB;
			return FPinConnectionResponse(ReplyBreakOutputs,
TEXT("Replace existing connections"));
		}
	}
```

```
	// Fall back to standard K2 rules
	return Super::DetermineConnectionResponseOfCompatibleTypedPins(PinA,
PinB, InputPin, OutputPin);
}

bool UAnimationGraphSchema::ArePinsCompatible(const UEdGraphPin* PinA,
const UEdGraphPin* PinB, const UClass* CallingContext, bool bIgnoreArray)
const
{
	// both are pose pin, but doesn't match type, then return false;
	if (IsPosePin(PinA->PinType) && IsPosePin(PinB->PinType) &&
IsLocalSpacePosePin(PinA->PinType) != IsLocalSpacePosePin(PinB->PinType))
	{
		return false;
	}

	// Disallow pose pins connecting to wildcards (apart from reroute
nodes)
	if(IsPosePin(PinA->PinType) && PinB->PinType.PinCategory ==
PC_Wildcard)
	{
		return Cast<UK2Node_Knot>(PinB->GetOwningNode()) != nullptr;
	}
	else if(IsPosePin(PinB->PinType) && PinA->PinType.PinCategory ==
PC_Wildcard)
	{
		return Cast<UK2Node_Knot>(PinA->GetOwningNode()) != nullptr;
	}

	return Super::ArePinsCompatible(PinA, PinB, CallingContext,
bIgnoreArray);
}

bool UAnimationGraphSchema::DoesSupportAnimNotifyActions() const
{
	// Don't offer notify items in anim graph
	return false;
}

void UAnimationGraphSchema::CreateFunctionGraphTerminators(UEdGraph& Graph,
UClass* Class) const
{
	if(const UAnimationGraphSchema* Schema =
ExactCast<UAnimationGraphSchema>(Graph.GetSchema()))
	{
		const FName GraphName = Graph.GetFName();

		// Get the function GUID from the most up-to-date class
		FGuid GraphGuid;

	FBlueprintEditorUtils::GetFunctionGuidFromClassByFieldName(FBlueprint
EditorUtils::GetMostUpToDateClass(Class), GraphName, GraphGuid);

		// Create a root node
		FGraphNodeCreator<UAnimGraphNode_Root> RootNodeCreator(Graph);
		UAnimGraphNode_Root* RootNode = RootNodeCreator.CreateNode();
		RootNodeCreator.Finalize();
```

```
            SetNodeMetaData(RootNode, FNodeMetadata::DefaultGraphNode);

            UFunction* InterfaceToImplement = FindUField<UFunction>(Class,
GraphName);
            if (InterfaceToImplement)
            {
                  // Propagate group from metadata
                  TArray<UAnimGraphNode_Root*> RootNodes;
                  Graph.GetNodesOfClass<UAnimGraphNode_Root>(RootNodes);

                  check(RootNodes.Num() == 1);
                  RootNodes[0]-
>Node.SetGroup(*FObjectEditorUtils::GetCategoryText(InterfaceToImplement).T
oString());

                  int32 CurrentPoseIndex = 0;
                  for (TFieldIterator<FProperty>
PropIt(InterfaceToImplement); PropIt && (PropIt->PropertyFlags & CPF_Parm);
++PropIt)
                  {
                        FProperty* Param = *PropIt;

                        const bool bIsFunctionInput = !Param-
>HasAnyPropertyFlags(CPF_OutParm) || Param-
>HasAnyPropertyFlags(CPF_ReferenceParm);

                        if (bIsFunctionInput)
                        {
                              FEdGraphPinType PinType;
                              if(Schema->ConvertPropertyToPinType(Param,
PinType))
                              {
                                    // Create linked input pose for each pose
pin type

      if(UAnimationGraphSchema::IsPosePin(PinType))
                                    {

      FGraphNodeCreator<UAnimGraphNode_LinkedInputPose>
LinkedInputNodeCreator(Graph);
                                    UAnimGraphNode_LinkedInputPose*
LinkedInputNode = LinkedInputNodeCreator.CreateNode();
                                    LinkedInputNode-
>FunctionReference.SetExternalMember(GraphName, Class, GraphGuid);
                                    LinkedInputNode->Node.Name = Param-
>GetFName();
                                    LinkedInputNode->InputPoseIndex =
CurrentPoseIndex;

                                    LinkedInputNode->ReconstructNode();
                                    SetNodeMetaData(LinkedInputNode,
FNodeMetadata::DefaultGraphNode);
                                    LinkedInputNodeCreator.Finalize();

                                    CurrentPoseIndex++;
                              }
                        }
                  }
```

```
                }
            }

            AutoArrangeInterfaceGraph(Graph);
        }
        else
        {
            Super::CreateFunctionGraphTerminators(Graph, Class);
        }
}


bool UAnimationGraphSchema::CanShowDataTooltipForPin(const UEdGraphPin&
Pin) const
{
    return !IsPosePin(Pin.PinType) &&
UEdGraphSchema_K2::CanShowDataTooltipForPin(Pin);
}

bool
UAnimationGraphSchema::CanGraphBeDropped(TSharedPtr<FEdGraphSchemaAction>
InAction) const
{
    if (!InAction.IsValid())
    {
        return false;
    }

    if (InAction->GetTypeId() ==
FEdGraphSchemaAction_K2Graph::StaticGetTypeId())
    {
        FEdGraphSchemaAction_K2Graph* FuncAction =
(FEdGraphSchemaAction_K2Graph*)InAction.Get();
        if (UAnimationGraph* AnimGraph =
Cast<UAnimationGraph>((UEdGraph*)FuncAction->EdGraph))
        {
            return true;
        }
    }

    return false;
}

FReply
UAnimationGraphSchema::BeginGraphDragAction(TSharedPtr<FEdGraphSchemaAction
> InAction, const FPointerEvent& MouseEvent) const
{
    if (!InAction.IsValid())
    {
        return FReply::Unhandled();
    }

    if (InAction->GetTypeId() ==
FEdGraphSchemaAction_K2Graph::StaticGetTypeId())
    {
        FEdGraphSchemaAction_K2Graph* FuncAction =
(FEdGraphSchemaAction_K2Graph*)InAction.Get();
```

```
            if (UAnimationGraph* AnimationLayerGraph =
Cast<UAnimationGraph>((UEdGraph*)FuncAction->EdGraph))
            {
                  if (UAnimBlueprint* TargetAnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForGraph(Animation
LayerGraph)))
                  {
                        return
FReply::Handled().BeginDragDrop(FAnimationLayerDragDropAction::New(InAction
, FuncAction->FuncName, TargetAnimBlueprint, AnimationLayerGraph));
                  }
            }
      }

      return FReply::Unhandled();
}

bool UAnimationGraphSchema::SearchForAutocastFunction(const
FEdGraphPinType& OutputPinType, const FEdGraphPinType& InputPinType, FName&
TargetFunction, /*out*/ UClass*& FunctionOwner) const
{
      TOptional<UEdGraphSchema_K2::FSearchForAutocastFunctionResults>
Result = SearchForAutocastFunction(OutputPinType, InputPinType);
      if (Result)
      {
            TargetFunction = Result->TargetFunction;
            FunctionOwner = Result->FunctionOwner;
            return true;
      }

      return false;
}

TOptional<UEdGraphSchema_K2::FSearchForAutocastFunctionResults>
UAnimationGraphSchema::SearchForAutocastFunction(const FEdGraphPinType&
OutputPinType, const FEdGraphPinType& InputPinType) const
{
      TOptional<UEdGraphSchema_K2::FSearchForAutocastFunctionResults>
Result;

      if (IsComponentSpacePosePin(OutputPinType) &&
IsLocalSpacePosePin(InputPinType))
      {
            // Insert a Component To LocalSpace conversion
            Result = UEdGraphSchema_K2::FSearchForAutocastFunctionResults{};
      }
      else if (IsLocalSpacePosePin(OutputPinType) &&
IsComponentSpacePosePin(InputPinType))
      {
            // Insert a Local To ComponentSpace conversion
            Result = UEdGraphSchema_K2::FSearchForAutocastFunctionResults{};
      }
      else
      {
            Result = Super::SearchForAutocastFunction(OutputPinType,
InputPinType);
      }
```

```
	return Result;
}

bool
UAnimationGraphSchema::CreateAutomaticConversionNodeAndConnections(UEdGraph
Pin* PinA, UEdGraphPin* PinB) const
{
	// Determine which pin is an input and which pin is an output
	UEdGraphPin* InputPin = NULL;
	UEdGraphPin* OutputPin = NULL;
	if (!CategorizePinsByDirection(PinA, PinB, /*out*/ InputPin, /*out*/
OutputPin))
	{
		return false;
	}

	// Look for animation specific conversion operations
	UK2Node* TemplateNode = NULL;
	if (IsComponentSpacePosePin(OutputPin->PinType) &&
IsLocalSpacePosePin(InputPin->PinType))
	{
		TemplateNode =
NewObject<UAnimGraphNode_ComponentToLocalSpace>();
	}
	else if (IsLocalSpacePosePin(OutputPin->PinType) &&
IsComponentSpacePosePin(InputPin->PinType))
	{
		TemplateNode =
NewObject<UAnimGraphNode_LocalToComponentSpace>();
	}

	// Spawn the conversion node if it's specific to animation
	if (TemplateNode != NULL)
	{
		UEdGraph* Graph = InputPin->GetOwningNode()->GetGraph();
		FVector2D AverageLocation =
CalculateAveragePositionBetweenNodes(InputPin, OutputPin);

		UK2Node* ConversionNode =
FEdGraphSchemaAction_K2NewNode::SpawnNodeFromTemplate<UK2Node>(Graph,
TemplateNode, AverageLocation);
		AutowireConversionNode(InputPin, OutputPin, ConversionNode);

		return true;
	}
	else
	{
		// Give the regular conversions a shot
		return Super::CreateAutomaticConversionNodeAndConnections(PinA,
PinB);
	}
}

bool IsAimOffsetBlendSpace(UBlendSpace* BlendSpace)
{
	return	BlendSpace->IsA(UAimOffsetBlendSpace::StaticClass()) ||
		BlendSpace->IsA(UAimOffsetBlendSpace1D::StaticClass());
}
```

```
void UAnimationGraphSchema::SpawnNodeFromAsset(UAnimationAsset* Asset,
const FVector2D& GraphPosition, UEdGraph* Graph, UEdGraphPin*
PinIfAvailable)
{
	check(Graph);
	check(Graph->GetSchema()->IsA(UAnimationGraphSchema::StaticClass()));
	check(Asset);

	UAnimBlueprint* AnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForGraph(Graph));

	const bool bSkelMatch =
UE::Anim::BP::Editor::IsSkeletonCompatible(AnimBlueprint, Asset);
	const bool bTypeMatch = (PinIfAvailable == nullptr) ||
UAnimationGraphSchema::IsLocalSpacePosePin(PinIfAvailable->PinType);
	const bool bDirectionMatch = (PinIfAvailable == nullptr) ||
(PinIfAvailable->Direction == EGPD_Input);

	if (bSkelMatch && bTypeMatch && bDirectionMatch)
	{
		FEdGraphSchemaAction_K2NewNode Action;

		UClass* NewNodeClass = GetNodeClassForAsset(Asset->GetClass());

		if (NewNodeClass)
		{
			check(NewNodeClass-
>IsChildOf(UAnimGraphNode_AssetPlayerBase::StaticClass()));

			UAnimGraphNode_AssetPlayerBase* NewNode =
NewObject<UAnimGraphNode_AssetPlayerBase>(GetTransientPackage(),
NewNodeClass);
			NewNode->SetAnimationAsset(Asset);
			NewNode->CopySettingsFromAnimationAsset(Asset);
			Action.NodeTemplate = NewNode;

			Action.PerformAction(Graph, PinIfAvailable,
GraphPosition);
		}
	}
}

void UAnimationGraphSchema::SpawnRigidBodyNodeFromAsset(UPhysicsAsset*
Asset, const FVector2D& GraphPosition, UEdGraph* Graph)
{
	check(Graph);
	check(Graph->GetSchema()->IsA(UAnimationGraphSchema::StaticClass()));
	check(Asset);

	FEdGraphSchemaAction_K2NewNode Action;

	UAnimGraphNode_RigidBody* NewNode =
NewObject<UAnimGraphNode_RigidBody>(GetTransientPackage());
	NewNode->Node.OverridePhysicsAsset = Asset;
	Action.NodeTemplate = NewNode;

	Action.PerformAction(Graph, nullptr, GraphPosition);
```

```
}

void UAnimationGraphSchema::UpdateNodeWithAsset(UK2Node* K2Node,
UAnimationAsset* Asset)
{
      if (Asset != NULL)
      {
            if (UAnimGraphNode_AssetPlayerBase* AssetPlayerNode =
Cast<UAnimGraphNode_AssetPlayerBase>(K2Node))
            {
                  if (AssetPlayerNode->SupportsAssetClass(Asset->GetClass())
!= EAnimAssetHandlerType::NotSupported)
                  {
                        const FScopedTransaction
Transaction(LOCTEXT("UpdateNodeWithAsset", "Updating Node with Asset"));
                        AssetPlayerNode->Modify();

                        AssetPlayerNode->SetAnimationAsset(Asset);

                        K2Node->GetSchema()->ForceVisualizationCacheClear();
                        K2Node->ReconstructNode();
                  }
            }
      }
}

void UAnimationGraphSchema::DroppedAssetsOnGraph( const TArray<FAssetData>&
Assets, const FVector2D& GraphPosition, UEdGraph* Graph ) const
{
      if (Graph != NULL)
      {
            if (UAnimationAsset* AnimationAsset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets))
            {
                  SpawnNodeFromAsset(AnimationAsset, GraphPosition, Graph,
NULL);
            }
            else if (UPhysicsAsset* PhysicsAsset =
FAssetData::GetFirstAsset<UPhysicsAsset>(Assets))
            {
                  SpawnRigidBodyNodeFromAsset(PhysicsAsset, GraphPosition,
Graph);
            }
      }
}

void UAnimationGraphSchema::DroppedAssetsOnNode(const TArray<FAssetData>&
Assets, const FVector2D& GraphPosition, UEdGraphNode* Node) const
{
      UAnimationAsset* Asset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets);
      UK2Node* K2Node = Cast<UK2Node>(Node);
      if ((Asset != NULL) && (K2Node!= NULL))
      {
            UpdateNodeWithAsset(K2Node, Asset);
      }
}
```

```
void UAnimationGraphSchema::DroppedAssetsOnPin(const TArray<FAssetData>&
Assets, const FVector2D& GraphPosition, UEdGraphPin* Pin) const
{
	UAnimationAsset* Asset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets);
	if ((Asset != NULL) && (Pin != NULL))
	{
		// Don't have access to bounding information for node, using
fixed offset that should work for most cases.
		const FVector2D FixedOffset(-250.0f, 50.0f);
		SpawnNodeFromAsset(Asset, GraphPosition + FixedOffset, Pin-
>GetOwningNode()->GetGraph(), Pin);
	}
}

void UAnimationGraphSchema::GetAssetsNodeHoverMessage(const
TArray<FAssetData>& Assets, const UEdGraphNode* HoverNode, FString&
OutTooltipText, bool& OutOkIcon) const
{
	UAnimationAsset* Asset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets);
	if ((Asset == NULL) || (HoverNode == NULL) || !HoverNode-
>IsA(UAnimGraphNode_Base::StaticClass()))
	{
		OutTooltipText = TEXT("");
		OutOkIcon = false;
		return;
	}

	bool bCanPlayAsset = SupportNodeClassForAsset(Asset->GetClass(),
HoverNode->GetClass());

	// this one only should happen when there is an Anim Blueprint
	UAnimBlueprint* AnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForNode(HoverNode)
);
	const bool bSkelMatch =
UE::Anim::BP::Editor::IsSkeletonCompatible(AnimBlueprint, Asset);

	if (!bSkelMatch)
	{
		OutOkIcon = false;
		OutTooltipText = LOCTEXT("SkeletonsNotCompatible", "Skeletons
are not compatible").ToString();
	}
	else if (bCanPlayAsset)
	{
		OutOkIcon = true;
		OutTooltipText =
FText::Format(LOCTEXT("AssetNodeHoverMessage_Success", "Change node to play
'{0}'"), FText::FromString(Asset->GetName())).ToString();
	}
	else
	{
		OutOkIcon = false;
		OutTooltipText =
FText::Format(LOCTEXT("AssetNodeHoverMessage_Fail", "Cannot play '{0}' on
this node type"),  FText::FromString(Asset->GetName())).ToString();
```

```
	}
}

void UAnimationGraphSchema::GetAssetsPinHoverMessage(const
TArray<FAssetData>& Assets, const UEdGraphPin* HoverPin, FString&
OutTooltipText, bool& OutOkIcon) const
{
	UAnimationAsset* Asset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets);
	if ((Asset == NULL) || (HoverPin == NULL))
	{
		OutTooltipText = TEXT("");
		OutOkIcon = false;
		return;
	}

	// this one only should happen when there is an Anim Blueprint
	UAnimBlueprint* AnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForNode(HoverPin-
>GetOwningNode()));

	const bool bSkelMatch =
UE::Anim::BP::Editor::IsSkeletonCompatible(AnimBlueprint, Asset);
	const bool bTypeMatch =
UAnimationGraphSchema::IsLocalSpacePosePin(HoverPin->PinType);
	const bool bDirectionMatch = HoverPin->Direction == EGPD_Input;

	if (bSkelMatch && bTypeMatch && bDirectionMatch)
	{
		OutOkIcon = true;
		OutTooltipText =
FText::Format(LOCTEXT("AssetPinHoverMessage_Success", "Play {0} and feed to
{1}"), FText::FromString(Asset->GetName()), FText::FromName(HoverPin-
>PinName)).ToString();
	}
	else
	{
		OutOkIcon = false;
		OutTooltipText = LOCTEXT("AssetPinHoverMessage_Fail", "Type or
direction mismatch; must be wired to a pose input").ToString();
	}
}

void UAnimationGraphSchema::GetAssetsGraphHoverMessage(const
TArray<FAssetData>& Assets, const UEdGraph* HoverGraph, FString&
OutTooltipText, bool& OutOkIcon) const
{
	if (UAnimationAsset* AnimationAsset =
FAssetData::GetFirstAsset<UAnimationAsset>(Assets))
	{
		UAnimBlueprint* AnimBlueprint =
Cast<UAnimBlueprint>(FBlueprintEditorUtils::FindBlueprintForGraph(HoverGrap
h));
		const bool bSkelMatch =
UE::Anim::BP::Editor::IsSkeletonCompatible(AnimBlueprint, AnimationAsset);
		if (!bSkelMatch)
		{
			OutOkIcon = false;
```

```
                if(AnimBlueprint && AnimBlueprint->bIsTemplate)
                {
                      OutTooltipText = LOCTEXT("TemplateNotAllowed",
"Template animation blueprints cannot reference assets").ToString();
                }
                else
                {
                      OutTooltipText = LOCTEXT("SkeletonsNotCompatible",
"Skeletons are not compatible").ToString();
                }
           }
           else if(UAnimMontage* Montage =
FAssetData::GetFirstAsset<UAnimMontage>(Assets))
           {
                OutOkIcon = false;
                OutTooltipText = LOCTEXT("NoMontagesInAnimGraphs",
"Montages cannot be used in animation graphs").ToString();
           }
           else
           {
                OutOkIcon = true;
                OutTooltipText = TEXT("");
           }
     }
     else if(UPhysicsAsset* PhysicsAsset =
FAssetData::GetFirstAsset<UPhysicsAsset>(Assets))
     {
           OutOkIcon = true;
           OutTooltipText = TEXT("");
     }
}

void UAnimationGraphSchema::GetContextMenuActions(UToolMenu* Menu,
UGraphNodeContextMenuContext* Context) const
{
     Super::GetContextMenuActions(Menu, Context);

     if (const UAnimGraphNode_Base* AnimGraphNode =
Cast<UAnimGraphNode_Base>(Context->Node))
     {
           {
                // Node contextual actions
                FToolMenuSection& Section = Menu-
>AddSection("AnimGraphSchemaNodeActions",
LOCTEXT("AnimNodeActionsMenuHeader", "Anim Node Actions"));
                if (GetDefault<UAnimBlueprintSettings>()-
>bAllowPoseWatches)
                {

     Section.AddMenuEntry(FAnimGraphCommands::Get().TogglePoseWatch);
                }

     Section.AddMenuEntry(FAnimGraphCommands::Get().HideUnboundPropertyPin
s);
           }

           if(Context->Pin && !IsPosePin(Context->Pin->PinType))
           {
```

```
                TSharedPtr<SWidget> BindingWidget =
MakeBindingWidgetForPin({ const_cast<UAnimGraphNode_Base*>(AnimGraphNode)
}, Context->Pin->GetFName(), false, true);
                if(BindingWidget.IsValid())
                {
                      FToolMenuSection& Section = Menu-
>AddSection("EdGraphSchemaPinActions");

     Section.AddEntry(FToolMenuEntry::InitWidget("BindingWidget",
BindingWidget.ToSharedRef(), LOCTEXT("BindingWidgetLabel", "Binding"),
true));
                }
           }
     }
}

void
UAnimationGraphSchema::HideUnboundPropertyPins(UAnimGraphNode_LinkedAnimGra
phBase* Node)
{
     TArrayView<FOptionalPinFromProperty> OptionalPins = Node-
>CustomPinProperties;

     for (FOptionalPinFromProperty& OptionalPin : OptionalPins)
     {
           FName PropertyName;
           FProperty* Property = nullptr;
           int32 PinIndex = 0;
           Node->GetPinBindingInfo(OptionalPin.PropertyName, PropertyName,
Property, PinIndex);
           if (Node-
>IsPinUnlinkedUnboundAndUnset(OptionalPin.PropertyName.ToString(),
EGPD_Input))
           {
                Node->SetCustomPinVisibility(false, PinIndex);
           }
     }
}

TSharedPtr<SWidget> UAnimationGraphSchema::MakeBindingWidgetForPin(const
TArray<UAnimGraphNode_Base*>& InAnimGraphNodes, FName InPinName, bool
bInOnGraphNode, TAttribute<bool> bInIsEnabled)
{
     const UAnimGraphNode_Base* FirstNode = InAnimGraphNodes[0];

     FProperty* PinProperty = nullptr;
     int32 OptionalPinIndex = INDEX_NONE;
     FName BindingName = NAME_None;
     if(FirstNode && FirstNode->GetPinBindingInfo(InPinName, BindingName,
PinProperty, OptionalPinIndex))
     {
           check(PinProperty);
           check(OptionalPinIndex != INDEX_NONE);
           check(BindingName != NAME_None);

           const bool bPropertyIsOnFNode = FirstNode->GetFNodeProperty() !=
nullptr && (FirstNode->GetFNodeProperty()->Struct->IsChildOf(PinProperty-
>GetOwner<UScriptStruct>()));
```

```
            UAnimGraphNode_Base::FAnimPropertyBindingWidgetArgs
BindingArgs(InAnimGraphNodes, PinProperty, InPinName, BindingName,
OptionalPinIndex);

            BindingArgs.OnGetOptionalPins =
UAnimGraphNode_Base::FAnimPropertyBindingWidgetArgs::FOnGetOptionalPins::Cr
eateLambda([bPropertyIsOnFNode](UAnimGraphNode_Base* InNode,
TArrayView<FOptionalPinFromProperty>& OutOptionalPins)
            {
                  if(UAnimGraphNode_CustomProperty* CustomProperty =
Cast<UAnimGraphNode_CustomProperty>(InNode))
                  {
                        if(bPropertyIsOnFNode)
                        {
                              OutOptionalPins = InNode->ShowPinForProperties;
                        }
```

ДОДАТОК В

Слайди презентації

Рисунок В.1 – Слайд 1 «Титульний»

Мета дослідження – аналіз та порівняння методів використання систем симуляції часток у сучасних ігрових движках (Unity, Unreal Engine, CryEngine та можливість їх використання в портативних пристроях.

Об'єкт дослідження – процеси створення, оптимізації та інтеграції VFX у ігрових проектах, а також їх вплив на графічну якість, продуктивність та іммерсивність гри.

2

Рисунок В.2 – Слайд 2 «Постановка задачі»

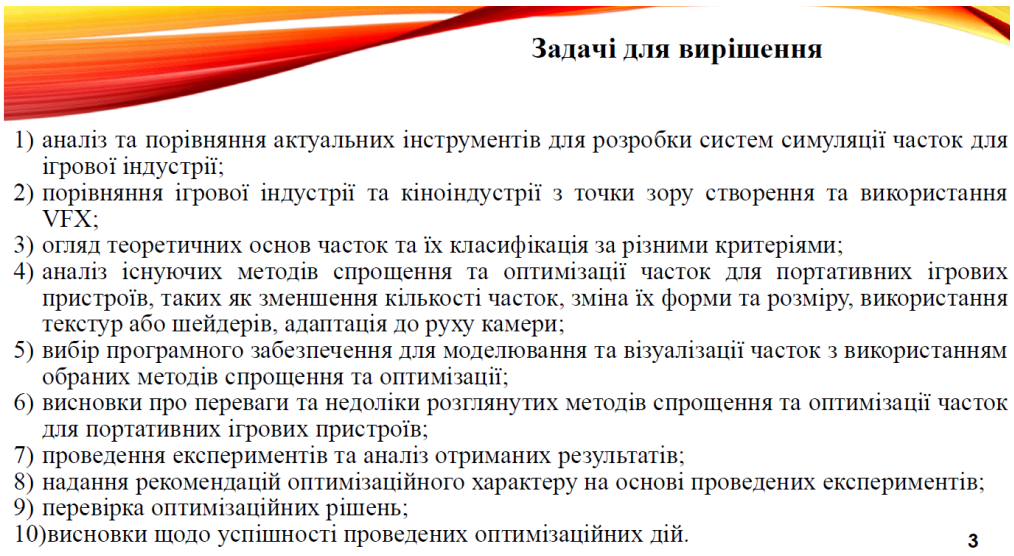

Рисунок В.3 – Слайд 3 «Задачі для вирішення»

Рисунок В.4 – Слайд 4 «Огляд та порівняння сучасних ігрових движків»

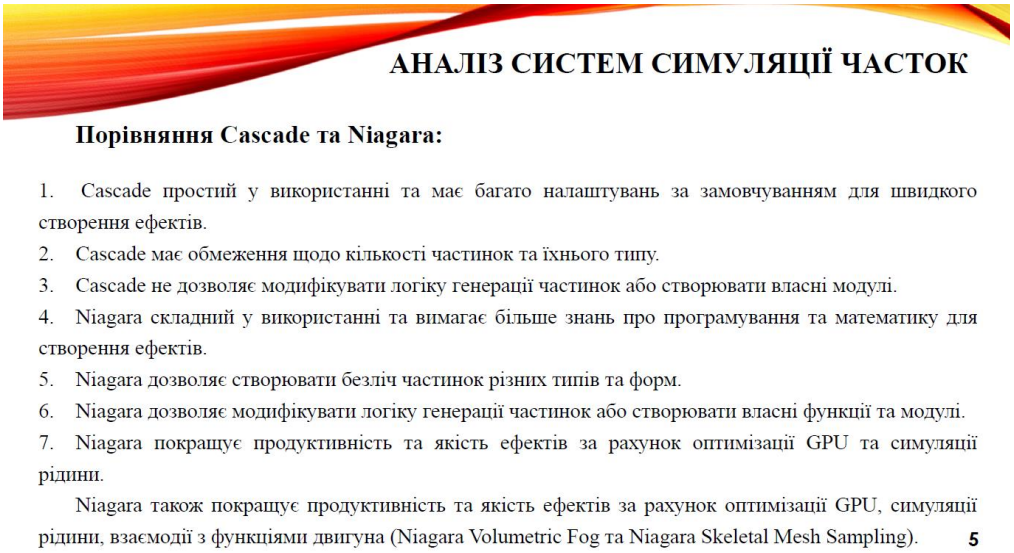

Рисунок В.5 – Слайд 5 «Аналіз систем симуляції часток»

Рисунок В.6 – Слайд 6 «Категорії модулів систем випромінювання часток»

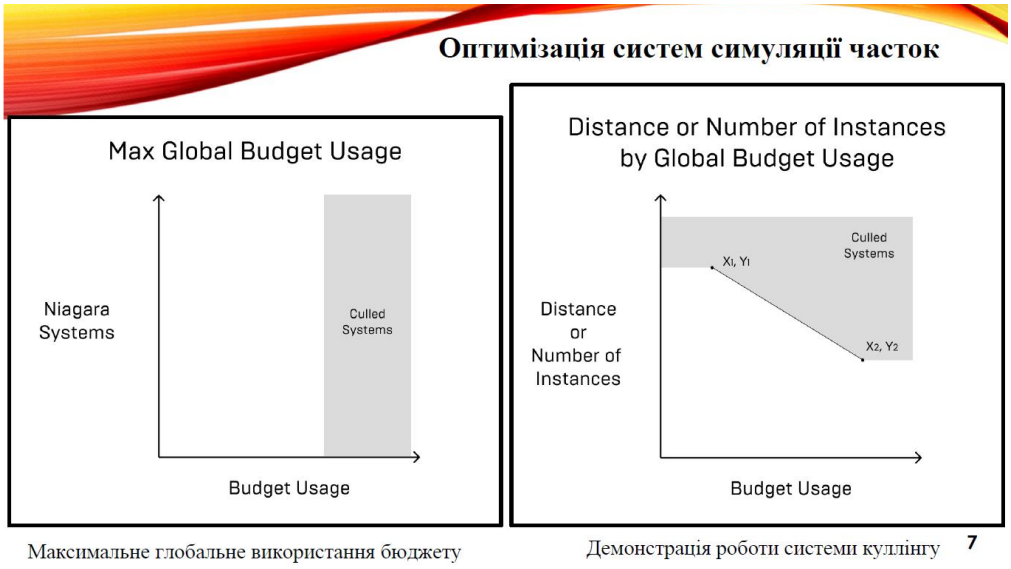

Рисунок В.7 – Слайд 7 «Оптимізація систем симуляції часток»

Рисунок В.8 – Слайд 8 «Експериментальні дослідження»

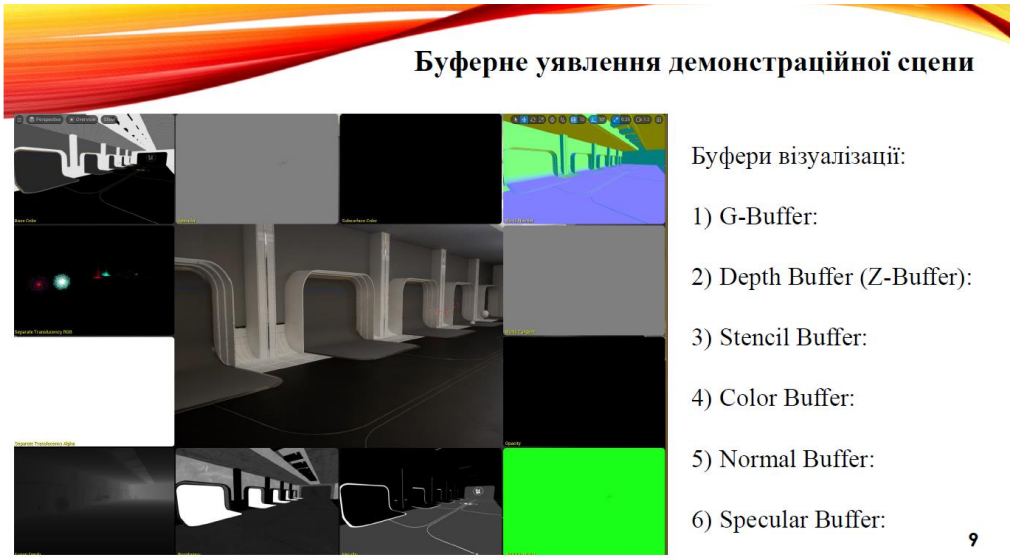

Рисунок В.9 – Слайд 9 «Буферне уявлення демонстраційної сцени»

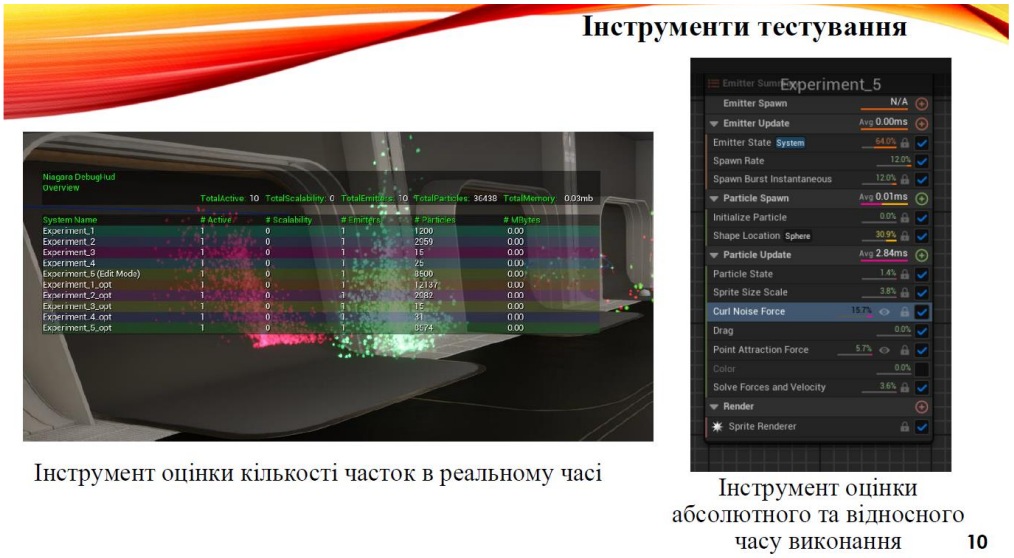

Рисунок В.10 – Слайд 10 «Інструменти тестування»

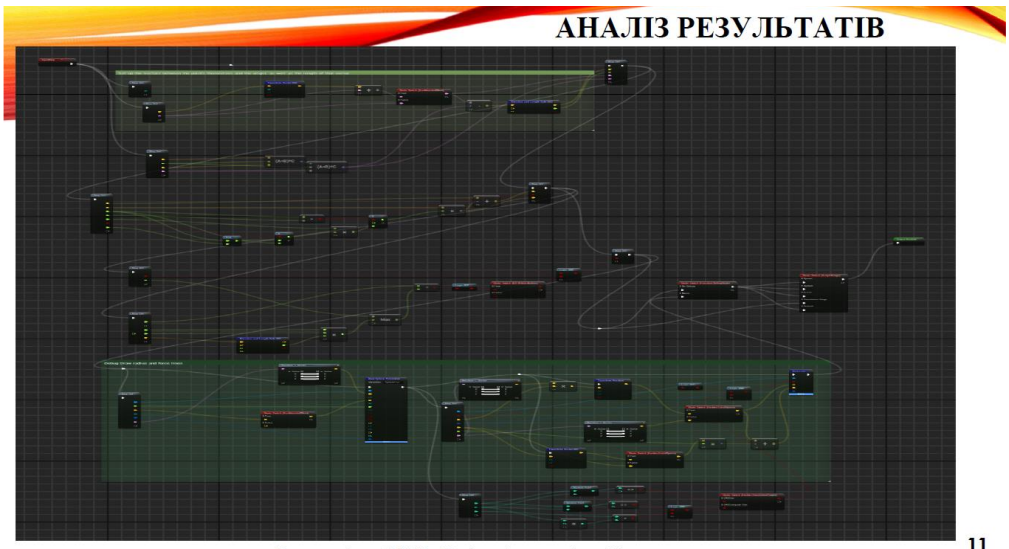

Рисунок В.11 – Слайд 11 «Аналіз результатів»

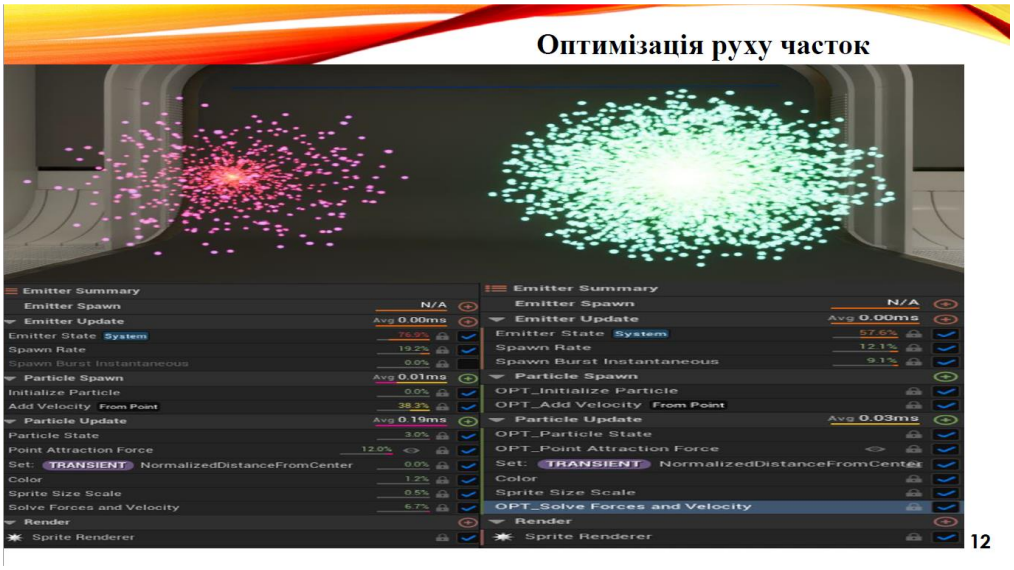

Рисунок В.12 – Слайд 12 «Оптимізація руху часток»

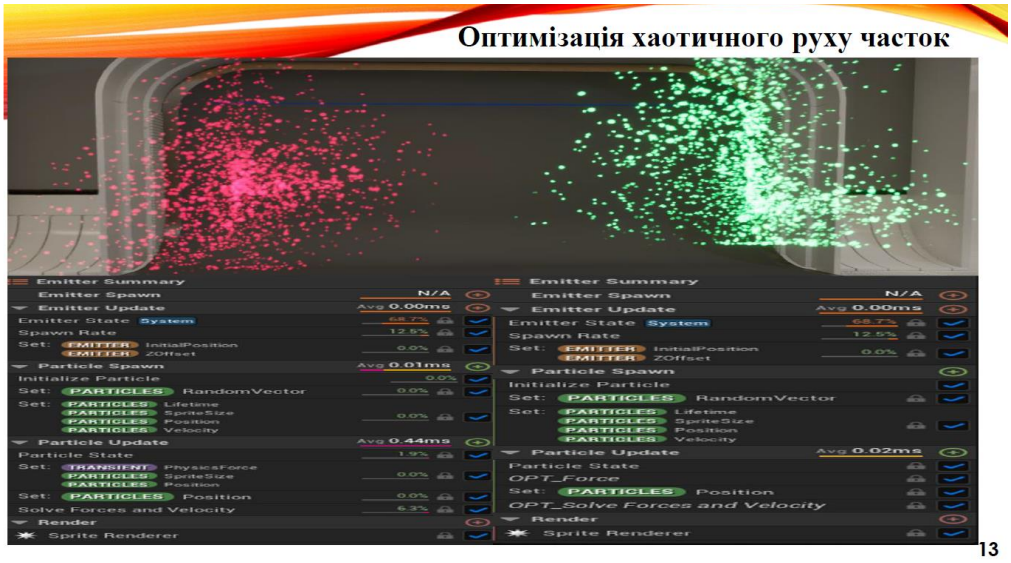

Рисунок В.13 – Слайд 13 «Оптимізація хаотичного руху часток»

Рисунок В.14 – Слайд 14 «Оптимізація воронкового руху часток»

Рисунок В.15 – Слайд 15 «Оптимізація колізії часток»

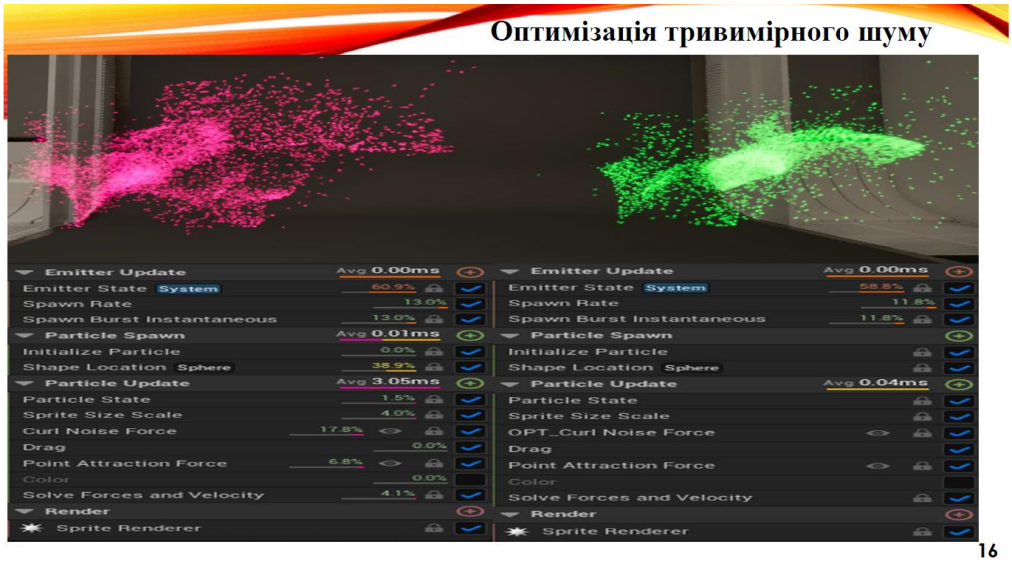

Рисунок В.16 – Слайд 16 «Оптимізація тривимірного шуму»

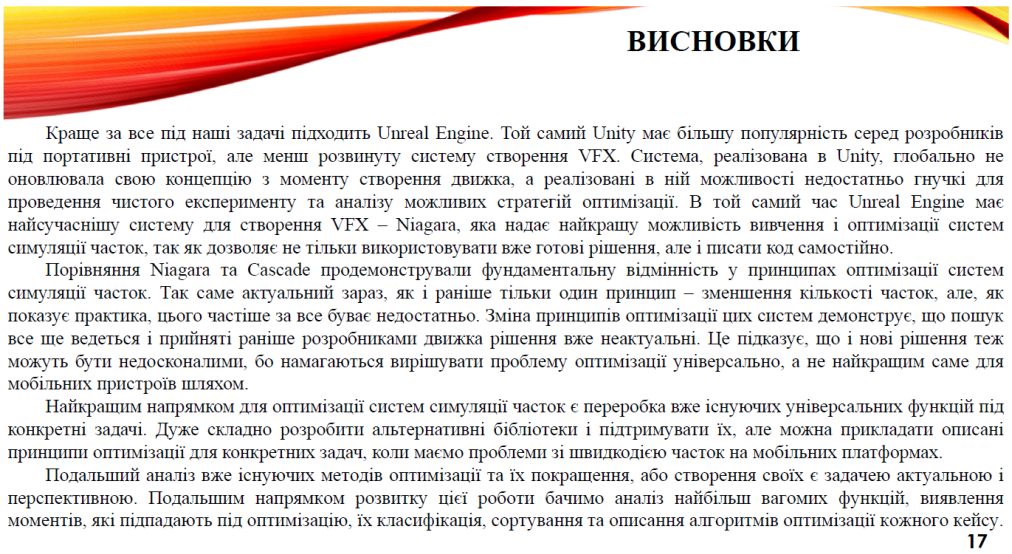

Рисунок В.17 – Слайд 17 «Висновки»

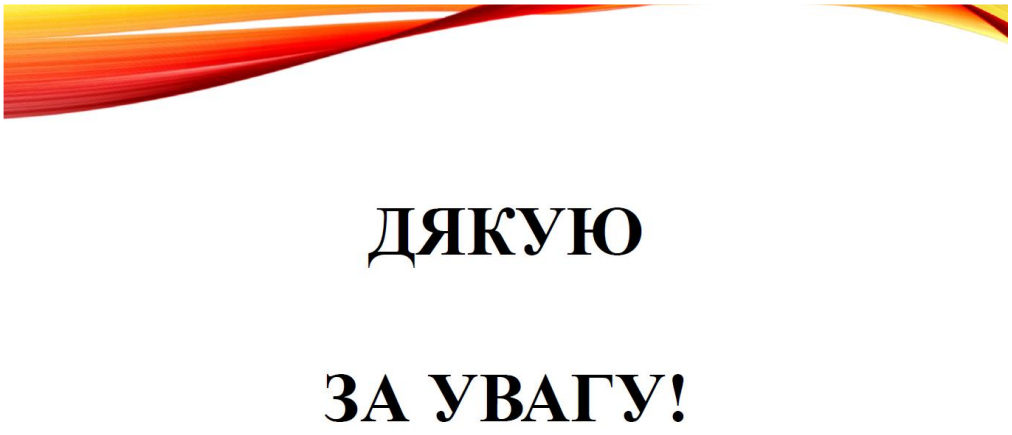

Рисунок В.18 – Слайд 18 «Дякую за увагу!»

ДОДАТОК Г

Апробація результатів роботи

RESEARCHED METHODS FOR SIMPLIFYING AND OPTIMIZING PARTICLES FOR PORTABLE GAMING DEVICES

Maksym Shulha[1], Dmytro Matvieiev[2], Oleksii Nazarov[3], Nataliia Nazarova[4]
[1]Master's student of the department of Software Engineering, NURE, Ukraine, maksym.shulha@nure.ua
[2]Assistant of the department of Software Engineering, NURE, Ukraine, dmytro.matvieiev@nure.ua,
ORCID iD: 0000-0002-0622-8159
[3]Associate professor of the department of Software Engineering, NURE, Ukraine, oleksii.nazarov1@nure.ua,
ORCID iD: 0000-0001-8682-5000
[4]Assistant of the Department of Higher Mathematics, NURE, Ukraine, nataliia.nazarova@nure.ua,
ORCID iD: 0009-0007-7816-7088

The object of the research is particle simulation systems and their modules. The aim of the work is to conduct a study of the performance of particle simulation systems and the optimization process to improve it. The methods of development and design are the analysis of the problem area of research, the choice of a particle simulation system based on their comparison for further study and optimization. As a result of work, a game simulation project was developed to demonstrate the effectiveness of the proposed optimization methods.
VERTICES, OPTIMIZATION, SIMULATION, PARTICLES SYSTEMS, SHADERS, INSTRUCTIONS COUNT, UNREAL ENGINE.

Об'єктом дослідження є системи симуляції часток та їх модулі. Метою роботи є проведення дослідження продуктивності систем симуляції часток та процес оптимізації для її покращення. Методами розробки та проектування є аналіз проблемної області дослідження, вибір системи симуляції часток на основі їх порівняння для подальшого вивчення та оптимізації. У результаті роботи було розроблено ігровий проект-симуляцію для демонстрації ефективності запропонованих методів оптимізації.
ВЕРТЕКСИ, ОПТИМІЗАЦІЯ, СИМУЛЯЦІЯ, СИСТЕМИ ЧАСТОК, ШЕЙДЕРИ, INSTRUCTIONS COUNT, UNREAL ENGINE.

1. Introduction

Visual effects (VFX) is a technology that allows you to create realistic effects such as fire, smoke, water, sparks, and more using a large number of small objects called particles.

These particles can have different properties, such as color, size, shape, speed, direction, etc., and interact with each other and the environment according to certain rules.

Particle simulation systems are widely used in modern game engines to create immersive visual effects that enhance the immersion and atmosphere of the game.

The purpose of the study is to analyze and compare the methods of using particle simulation systems in modern game engines such as Unity, Unreal Engine, CryEngine and the possibility of using them in portable devices.

The study focuses on the processes of creating, optimizing and integrating VFX in game projects, as well as their impact on the graphical quality, performance and immersiveness of the game.

The study aims to identify the advantages and disadvantages of different approaches to VFX implementation and provide recommendations for their effective use.

In this article, we will review the basic principles and methods of particle simulation systems, as well as examples of their application in various game genres. We will also analyze the advantages and disadvantages of this technology, as well as the possibilities for its further development and improvement.

2. Description of subject area

The proliferation of tools and technologies for MPE development is reflected in the computer game industry [1], interactive cinema, and augmented and virtual reality applications.

As development tools evolve, the need for dynamic model optimization comes to the fore. This is due to the fact that the simplification of the process of developing rather complex models has led to a decrease in developers' interest in multifactor optimization of existing models - it is easier to build a new model than to optimize an existing one.

The optimization process can be applied to almost all elements of modern engines. This is due to the fact that the engines provide as much functionality as possible to ensure compliance in the applications they are designed for. For applications such as the film industry or simulations, where models do not run in real time, optimization is not critical because it only affects the comfort of working with the project, not the final result, which can go through the rendering process at any time. The main area where optimization is critical is in real-time software applications. For this class of applications, the main requirement for the optimization process is to minimize optimization-related delays and maximize synchronization of all used engine elements, since they can run on different hardware and, in the future, be scaled to new portable devices. This fact attracts special attention because mobility is developing quite actively, and therefore optimizing existing applications for new types of mobile devices becomes more important than ever.

Рисунок Г.1 – Перша сторінка статті в журналі «Біоніка інтелекту»

The optimization process remains complex and situational in its parameters, and includes many criteria that require an individual approach. Modern engines consist of modules that work together and can influence each other. Due to this fact, it is often impossible to disable them: for example, games adapted for virtual reality rarely use a standard interface module, which requires a lot of resources for initialization and computation. Also, the development team often does not include experienced specialists who can understand the structure of the engine code and disable a particular module in a way that does not affect other modules. In some engines, such as Unity, the engine code is not editable, requiring optimization at a higher level.

3. Statement of research problem

The article highlights the following tasks to be considered:

1) analysis and comparison of current tools for developing particle simulation systems for the game industry;

2) to compare the gaming industry with the film industry in terms of the creation and use of VFX;

3) review the theoretical foundations of particles and their classification according to various criteria;

4) analysis of existing methods for simplifying and optimizing particles for handheld gaming devices, such as reducing the number of particles, changing their shape and size, using textures or shaders, adapting to camera movement;

5) select software for modeling and visualizing particles using the selected simplification and optimization methods;

6) conclusions about the advantages and disadvantages of the considered methods of particle simplification and optimization for portable gaming devices.

This set of questions should be sufficient for a first overview of the industry, the VFX development tools [2] and the problems to be solved. The results will also help to adjust the process of analyzing the results and the methods that will be used to solve problems with the optimization of particle simulation systems.

4. Research Facilities

The game engines listed in the introduction will be used for the research. In the course of this research, we will analyze the existing modern engines, review their pros and cons, and then choose the one that suits us best.

Since we are focusing on portable devices, the most important parameter of these engines will be the ability to build projects for mobile devices using modern libraries.

Analyzing optimization for consoles is quite complicated, because it requires the presence of so-called "devkits" - consoles sent by their owners for development. This is impossible for the learning process. There are also personal computers, but the variability of their hardware is very large, which makes it impossible to adequately evaluate the results of optimization, as video cards of different generations have fundamental differences.

That's why a personal computer will be used to test intermediate results to speed up the analysis process to obtain relative results and formulate optimization methods. Further testing will be done on a modern smartphone, as this platform is available and meets the hardware limitations that can reflect the effectiveness of the optimization methods found.

5. Review and compare modern game engines

Choosing the right game engine is by far the most important decision a game developer has to make to get the best results from their product. The first puzzle a game developer has to solve when creating a game is which game engine to use to get the best user experience. A game engine helps to create not only classic arcade games [3] like Ping, Tetris, Snake, but also innovative and advanced leveling games like GTA and Assassin's Creed.

Every game developer has heard of some incredible game engines like Unreal Engine, CryEngine and Unity3d. These are some of the most popular and leading game engines for creating advanced games today. Each of these 3 game engines has its own qualities and potential. We need to be clear about the nature of our project, for example, license budget, game platform, dimensions (2D or 3D), and so on. So your goal should be clear before choosing the best game engine 2023 for our project.

Unity has a large number of contributors in its community who can help us with the project right away. Another feature that makes it stand out as one of the best 3D game engines in 2023 is its ability to support a number of file formats used in leading 3D programs, including 3D Max, Blender, CINEMA, Maya, Softimage, and many others. In addition, game developers have access to more than 15,000 free and paid 3D models, audio, animations, editor extensions, materials, scripts, and shaders for use in game development.

Unity 3D uses C# or JavaScript, which is more desirable than C++ because you don't have the hassle of switching from Java to C# compared to C++. Nevertheless, Unity 3D has a simple and fast interface and is light enough to run even on Windows XP with Service Pack 2 (SP2).

Second on the list is Unreal Engine 4, which is the latest engine released by one of the largest American video game and software development companies, Epic Games. Unreal Engine 4 is the successor of the Unreal Development Kit, commonly known as UDK in the gaming world.

Unreal Engine 4 offers incredible graphics that add a realistic touch to the gaming experience with features such as advanced dynamic lighting. What makes this game engine even more amazing is its new particle system, which has the ability to handle up to a million particles in a single scene.

In addition, UE4 is completely free to use, but you must pay a royalty of 5% of the money you make from your Unreal Engine 4 games. In short, Epic Games gets 5% of everything you make, whether it's in-app purchases, in-game advertising, or money you charge users to buy your game. However, the creators of Unreal Engine 4 allow developers to use the full

Рисунок Г.2 – Друга сторінка статті в журналі «Біоніка інтелекту»

version of Unreal Engine 4 for free if the revenue you make from your game is up to $3,000 per quarter.

In addition, Unreal Engine 4 uses Blueprint Visual Scripting technology, which allows you to create games using Blueprint. However, such games have some limitations. Among other things, the bad thing about this engine is that it is not capable of developing games for last generation consoles.

Last but not least on our list of the best game engines in 2023 is CryEngine. First introduced by major development company Crytek in the first Far Cry game, CryEngine is undoubtedly one of the most powerful and dominant game engines we have today. What makes CryEngine worthy of the list is its graphical capabilities, which eclipse those of Unity and are equivalent to what Unreal has. Although CryEngine is a heavy and powerful game engine, it takes a little time for the user to be able to use this platform effectively and is a little harder to understand for beginners who have not used other game engines before.

The CryEngine supports virtual reality and has amazing visual effects, including a three-dimensional fog and cloud visualization system that gives clouds a full 3D spatial visualization and realistic rendering for fog and weather effects. Furthermore, the best part of choosing the CryEngine platform for game development is that it does not require its users to pay royalties during game development. However, to get access to CryEngine, you need to pay a fixed amount, namely $9.90/- per month. In addition, CryEngine has a dedicated Q&A forum called CryEngine Answers that clears all your doubts and queries and helps you to have the best experience.

While comparing Unreal, Unity and CryEngine, we came across the best features that these game engines offer us. Comparing the performance of Unity and Unreal, we realized that Unity is a better platform for developing mobile and 2D/3D games, while Unreal is best suited for developing highly graphical and photorealistic games. This is the main difference between Unreal and Unity.

CryEngine, on the other hand, also has the ability to create games with high graphics. Furthermore, when comparing CryEngine to Unreal 2023 on the scale of providing next-generation platform features with a more attractive pricing model, CryEngine undoubtedly outperforms Unreal with its cost-effective structure. However, for a beginner, when it comes to CryEngine 5 vs. Unreal Engine 4 based on lightness and minimalism, despite the amazing features that CryEngine has, Unreal Engine 4 and Unity are definitely worth a try.

At the end of this analysis we came to the conclusion that the most versatile and high-quality particle simulation system is implemented in Unreal Engine. Moreover, both systems of this engine have developed tools for optimization, not only for the creation of these particles. Therefore, further analysis should be done on this engine.

6. Industry analysis and Unreal Engine 4-5's place in it

The late 90's saw a monumental development in personal computers and the games that could be played on them. Graphics improved rapidly, and although ridiculous by today's standards, the development of 3D first-person graphics made video games much more immersive. The Internet was rapidly spreading to consumers, making it easier to participate in online games, join discussions on gaming forums, and even learn how to program games yourself. And the games released during this time are titles we still talk about with reverence: Doom, Sim City, Duke Nukem, Half-Life, StarCraft, Myst, and many others (see Figure 1).

Fig.1. Doom

But Unreal Tournament was in a league of its own: fast paced, large maps, and AI computer opponents that, for the first time, actually seemed intelligent. A community of players began to grow around the game, modifying it, making it their own, and sharing their creations with the community. All of this happened thanks to a game development company with an extremely bright future and the software they created called Unreal Engine.

Founded in 1991, Epic Games is one of the few software company success stories that has earned a rightful place in gaming history. Their first few games were commercially successful, but what really put them on the map came at the end of the millennium when they released Unreal[4] in 1998, followed by its sequel, Unreal Tournament, the following year.

The game's gameplay, graphics, and features set it apart from many other first-person shooters on the market. But beyond the game itself, Epic Games made a decision that made the game legendary. They decided to ship the game with the same tool - the Unreal Editor - that is used to create levels and gameplay, allowing players to customize the game and make it their own. It's one of many game engines developed over the years, but it's far from the most successful or widely used.

"If you think of a technology that has stood the test of time, Unreal Engine is one of the biggest," says Jacob Feldman, a software and rendering solutions engineer at CoreWeave who moonlights as an authorized Unreal Engine instructor. "It has over 20 years of code, so it's a significant, complex product."

Рисунок Г.3 – Третя сторінка статті в журналі «Біоніка інтелекту»

Realizing that they had a winner with the Unreal Engine, the Epic team began licensing the software so that other developers could build their own games on top of it. Fast forward to 2017, when the studio released Fortnite, a cultural sensation that would further cement the legacy of Epic and the Unreal Engine.

More recently, the Unreal Engine has played a major role in another cultural moment that has propelled the tool into an entirely new industry: movie production [5] (see Figure 2).

Fig.2. Using UE4 in cinema

In 2019, Disney released the award-winning series The Mandalorian as part of the lineup for its new streaming service, Disney+. The show became an instant hit, continuing the long history of Star Wars productions that have pushed the technological boundaries of the film industry. "The Mandalorian" demonstrated a giant leap forward in set design by using real-time set rendering with the Unreal Engine, effectively making green screen obsolete.

"Unreal made a leap into broader industries like film after Fortnite came out," says Feldman. "It became clear that a game engine like Unreal, designed for flexibility, performance, and visual quality, had more applications than just gaming. This became especially clear as more powerful hardware began to enable things like real-time ray tracing and other visual effects that were simply not possible 5-10 years ago. The visual effects industry has some of the most stringent requirements of any visual medium, so Epic has made a serious commitment to supporting the kinds of tools that VFX studios need.

Filmmakers have always found ways to trick the audience's eyes into thinking they are seeing something larger than what is actually there - what the industry calls "set extension. In the early days of Hollywood, this involved literally drawing scenes that were then seamlessly integrated into the set to create the effect of infinite space.

Later, green-screen technology became the preferred method for creating large worlds. However, there are some significant problems with this method. It can be difficult for actors and crew on set to perform convincingly against an empty, bright green background. The actors cannot see the world around them (see Figure 3) and therefore cannot react to it in a way that is convincing to the viewer.

But what if there was a way to create large worlds on command that actors could see right in front of them? It would be almost like throwing those actors into a video game and having them perform in that world, with all the views available to their own eyes.

Fig.3. Shooting a movie

Mandalorian creator Jon Favreau approached the team at Epic Games to see if they could help him do this, essentially immersing his actors in a video game. Using massive high-definition LED screens surrounding their set, the Mandalorian team took the old Hollywood concept of painted sets and brought it into the twenty-first century. Now these painted sets were living, breathing environments with dynamic lighting and movement. The actors could see them and react to them. The physical elements in the foreground blend seamlessly into the virtual background.

But it gets even better-using game engine technology, you can tie camera movement to the background (see Figure 4), allowing it to change with the camera to perfectly match the movements and provide smooth parallax movement that is indistinguishable from the real shot.

Fig.4. The process of filming «Mandalorian»

In essence, they created the real-time graphics that underpin movie production. There is no need to clean up in post-production; these effects are done in the camera in real time.

"It's a huge help for the actors and the crew to be able to see their surroundings as they happen," says Feldman. "Today, an actor has to imagine what is happening around them on a green screen. This advancement allows them to see it."

In addition to making life easier for the actors, the real-time environment also makes the work of the post-production team infinitely less stressful. On the green screen, reflection is a constant battle. Any shiny material will reflect the green environment and ruin the VFX effect. These reflections must be removed, and then virtual light must be added to the tiny reflections in glasses, helmets, or pieces of metal in the environment. Real-time environments eliminate all of this because the reflections captured by the camera reflect the world as it should be in the story.

Рисунок Г.4 – Четверта сторінка статті в журналі «Біоніка інтелекту»

Epic Games took the wraps off its latest game engine, Unreal Engine 5, and pretty much broke the gaming-interested part of the Internet. Running on the PlayStation 5 developer hardware version, the results uploaded to Vimeo looked pretty stunning at first, but when you realized they were real-time gameplay footage, they were truly impressive.

The visual quality is virtually unmatched by anything other than today's high-end pre-processed visual effects, with an incredible level of detail and truly photorealistic lighting [6]. What's even better, of course, is that the news about UE5 has lit up the broadcast graphics, movie effects, virtual studio, and pretty much anywhere else that high quality visual effects are used.

While other engines like Unity are available, Epic has done a nice job of combining a leading feature set with ease of use and a business model that encourages the use of UE in other software. For example, any game developer using Unreal commercially pays no licensing fees until gross software revenues reach $1 million (recently raised from $50,000), and this has helped to drive its widespread integration across a range of technologies. For non-game developers, it is 100% royalty free.

Its use in the graphics stack is similar to how connectivity to the IT industry as a whole has accelerated development in the broadcast sector; it allows a relatively small industry to leverage the development efforts of a much larger one, and the speed of change we're seeing as a result is impressive.

"When the new Unreal Engine 5 comes out, you won't be able to tell what's real and what's not," said Phil Ventre, vice president of sports and broadcast at Ncam Technologies. "The integration of game engines is democratizing the way companies use AR and VR, and it won't just be a tier-one technology for broadcasters in the future."

While the new demo was created in part to showcase some of the very smart new technologies in the upcoming PS5, such as the literally game-changing M.2 SSD, UE5 showcases some new technologies that are dramatically moving the goalposts for real-time CG work. Two in particular are worth mentioning: Nanite and Lumen.

Nanite is a new virtualized geometry[7] of micropolygons that essentially allows artists to create as much geometric detail as they want. It's translatable and scalable in real time, which is important to consider when you're planning an engine that will run on everything down to a smartphone (see Figure 5).

Then there is Lumen. This is a fully dynamic global lighting system that responds instantly to changes in scene and lighting, and is part of the secret to why the game's graphics look so good on a console. It's capable of reproducing diffuse bounce reflections with infinite bounce and indirect specular reflections in what Epic calls "vast, detailed environments on scales from kilometers to millimeters. It's also adaptable: Punch a hole in a wall and the scene will change to accommodate the light coming in through the hole.

Fig.5. Demo scene on UE5

"The ability to light and render hundreds of millions of polygons in real time is a quantum shift that will change the way filmmakers interact with the images they create," said Miles Perkins, business development manager at Epic Games. "These new technologies will allow creatives to see the fullness of their vision without having to separate different parts of their shots, viewing animation separately from lighting, separately from environments and effects. Everything will be right in front of them, fully controllable. Filmmakers will be able to compose and light shots in real time, whether they are physical, virtual, or a combination of the two."

Perhaps one of the key points here is that the Unreal Engine, currently running at version 4.25, is already very good.

"We are currently using Unreal Engine 4 extensively in both pre-production and virtual production," said Hugh MacDonald, director of technology innovation at Nviz. "For pre-production, the real-time nature of UE4 means we can get incredibly high quality images with minimal rendering time. We're also using it for virtual production, which allows for better integration than we could have done in the past.

Nviz uses Unreal in two main tools that illustrate how it is being used for pre-production work in the industry and where it can go. The virtual camera system allows for virtual scouting of pre-production environments and allows directors and cinematographers to get hands-on experience with the camera, while the simulcast toolkit is tightly integrated with Unreal and allows the crew on set to preview what the shot will look like after visual effects are added in post.

"Unreal allows us to ensure that it's both flexible and high quality," says MacDonald. "From what we know of UE5 so far, the main leaps will be in geometry detail, as much higher resolution assets can be used and rendered. This is a much better fit for the movie VFX workflow, as it is hoped that assets will require less editing to make them engine ready. Fully dynamic lighting, which is lumen, will mean that there will be less need for lighting baking to get the same result. This will allow us to keep scenes fully dynamic, allowing us to adjust lighting in real time during production if necessary, which is often requested on set as physical lighting changes from shot to shot."

New Creativity Along with the increase in quality, which Ventre of NCam compares to the leap from UE3 to UE4, UE5 opens up the tantalizing prospect of introducing both new ways of working and new ways of creatively exploring virtual spaces.

Рисунок Г.5 – П’ята сторінка статті в журналі «Біоніка інтелекту»

"The Unreal Engine is going to be a big part of the future of cinema," says CVP's Sam Mir. "It brings back the ability to get practical in-camera effects, whether it's interactive lighting on an actor or dynamic background changes in real time, even if they were created in a virtual space. The ability to get instant feedback on how something is going to look is invaluable."

This will find its way into many more living spaces than before. McDonald mentions theater and events, where video screens have become part of the interaction with lighting to create entirely new live spectacles, while virtual sets will take another leap in quality to become so indistinguishable from the real thing that only a live audience will influence the decision to use a physical set. Even these viewers will be able to see completely different shows, with mixed elements of augmented reality seamlessly integrated into the final TX. In the post-covid era, you can easily get three guests on the couch for a chat show from the comfort of your own home, and no one will be the wiser.

And, of course, there's the possibility that this could accelerate the development of live shots on LED screens, as pioneered by shows like The Mandalorian.

"While Unreal has been used a few times on LED screens during shoots, the new updates will hopefully take it much further and get a higher percentage of finished shots straight from the camera," MacDonald enthuses.

7. Particle simulation systems available in UE5

Cascade and Niagara[8] are two particle systems used in Unreal Engine to create effects such as fire, smoke, rain, snow, and sparks. Cascade is an older system introduced in Unreal Engine 3, while Niagara is a newer system added in Unreal Engine 4.25. Both systems have their advantages and disadvantages, which will be discussed below.

Cascade is based on the concept of emitters and modules. An emitter is an object that generates particles according to certain rules, and a module is a component that changes the properties of the particles, such as color, size, speed, rotation, etc. Cascade allows you to create complex effects from multiple emitters and modules and customize them using a graphical interface. Cascade also supports features such as GPU acceleration, particle collisions, lighting and shadow interaction, sprite animation, and more.

Niagara is based on the concept of systems and emitters. A system is an object that manages one or more emitters and their logic. An emitter is an object that generates particles according to certain rules, but in Niagara these rules can be specified using graph editors or scripting languages. Niagara allows you to create more flexible and dynamic effects with a variety of particle types, including mesh particles, line particles, ribbon particles, and more. Niagara also improves the performance and quality of effects through GPU optimization, fluid simulation, and interaction with engine functions.

A comparison of Cascade and Niagara can be seen as follows:

1) Cascade is easy to use and has many default settings for creating effects quickly.

2) Cascade has limitations on the number and type of particles.

3) Cascade does not allow you to change the logic of particle generation or create your own modules.

4) Niagara is difficult to use and requires more programming and math skills to create effects.

5) Niagara allows you to create many different types and shapes of particles.

6) Niagara allows you to modify the logic of particle generation or create your own functions and modules.

7) Niagara improves performance and effect quality through GPU optimization and fluid simulation.

8. Cascade

The basic and overarching concept of the cascade is that of modular particle systems. In some 3D effects packages, such as Maya, a particle system is created with most of the necessary properties for the behavior. The user then modifies these properties to achieve the desired result.

In Cascade, on the other hand, the particle system starts with just a few basic properties and a few behavior modules.

Each module represents a specific aspect of particle behavior and contains only the properties that control that behavior, such as color, birth position, motion, scaling, and many others. The user can then add or remove modules as needed to further define the behavior of the particles. Since only the modules for the desired behavior are added, there is no unnecessary computation of unnecessary properties.

Best of all, modules can be easily added, removed, copied, and even tried on emitters in the particle system, making complex setups very easy to achieve once the user is familiar with the available modules.

Some modules come standard with a particle emitter. When a new sprite emitter [9] - a key component of any particle system - is added to a particle system, it is always created with the following default modules:

1) Required - contains a variety of properties that are absolutely necessary for the particle system, such as the material applied to the particles, how long the emitter should emit particles, and many others.

2) Spawn - this module controls how fast the particles will appear from the emitter, whether they will appear in bursts, and any properties related to the particle birth time.

3) Lifetime - this parameter determines how long each particle will live after its birth. Without this module, particles will live indefinitely.

4) Initial Size - controls the size of the particle at the moment of its birth.

Рисунок Г.6 – Шоста сторінка статті в журналі «Біоніка інтелекту»

5) Initial Velocity - controls the movement of the particle at the moment of its birth.

6) Color Over Life - this module controls how the color of each particle will change during its life.

The Required and Spawn modules are permanent and cannot be removed from the emitter. All other modules can be removed at will.

Particle systems are also very closely related to the different materials and textures applied to each particle. The main task of the particle system itself is to control the behavior of the particles, while the specific look and feel of the particle system as a whole is often controlled by the materials.

There are many modules that can be added to particle emitters. To avoid confusion, these modules are divided into different categories. These categories include those listed in Table 1.

Table 1. Categories of particle emission system modules

Category	Description
Acceleration	Modules that control how the acceleration of particles can be affected by, for example, drag forces.
Attraction	Modules that control particle movement by attracting particles to different points in space.
Camera	Modules that control the movement of particles in camera space, allowing the user to make them appear closer or farther away from the camera.
Collision	Modules that control how collisions between particles and geometry are handled.
Color	Modules that control the color of particles.
Event	Modules that control the activation of particle events, which in turn can trigger a variety of in-game reactions.
Kill	Modules that control the removal of particles.
Lifetime	Modules that control how long particles live.
Light	Modules that control the light emitted by particles.
Location	Modules that control the birthplace of particles relative to the location of the Actor emitter.
Material	Modules that control the material of the particles themselves.
Orbit	Modules that provide orbital behavior of the screen space to add movement to the effects.
Orientation	Modules that allow you to set the rotation axis of the particles.
Parameter	Modules that can be parameterized or controlled by external sources such as blueprints and wounds.
Rotation	Modules that control the rotation of particles.
RotationRate	Modules that control the change of rotation speed, such as spin.
Size	Modules that control the size of particles.
Spawn	Modules that add special particle appearance rates, such as spawning particles based on distance moved.
SubUV	Modules to display animated sprite sheets on a particle.
Velocity	Modules controlling each particle's speed.

Two important concepts to keep in mind when working with particle modules are initial and lifetime. Initial modules are used to control some aspect of the particle at the moment of its birth. The Over Life or Per Life modules are used to allow some aspect of the particle to change during its lifetime.

For example, the Initial Color module allows you to set the color of the module at the time of birth, while the Over Life property allows the color of the particle to gradually change between the time of birth and the time of death.

If you change the property to a type of distribution that changes over time, some modules use "relative time" and some use "absolute time".

Absolute time [10] is essentially the time that the emitter contains. If you have an emitter set up for 3 cycles of 2 seconds, the absolute time for the modules in that emitter will go from 0 to 2 seconds three times.

The relative time is between 0 and 1 and indicates the lifetime of each particle.

As you work with Cascade to create your own particle effects, it's important to keep in mind how each object is related to the others. We've already discussed the concept of modules in this document, but they are only one component of a complete particle effect. In general, the components of a particle system are modules, emitters, particle systems, and emitter actors.

Just as there are many types of effects you'll want to create with your particles, there are also many types of emitters to help you create exactly what you need. Below is a list of the available emitter types:

Note that all emitters, regardless of type, are sprite emitters by default. Various Emitter Type Data modules are then added to the emitter to change its type to something else.

Not all aspects of a particle system can be defined in advance. Sometimes certain parts of the particle system's behavior need to be controlled or changed at runtime. For example, you may want to create a magic effect that changes color based on the amount of magic energy consumed during the spell. In such cases, you'll need to add parameters to the particle system.

A parameter is a type of property that can send or receive data to/from other systems, such as Blueprints, Matinee, Material, or many other sources. In Cascade, a parameter can be assigned almost any property, meaning that the property can be controlled from outside the particle system.

For example, setting a parameter to control the rate of creation of a fire effect can allow the player to increase or decrease the amount of flame emitted during operation.

Conversely, there are parameter modules that can be added to a particle system that in turn can control other things in the level, such as the color used in a particular material.

Рисунок Г.7 – Сьома сторінка статті в журналі «Біоніка інтелекту»

Particle systems can easily become very expensive to compute. Even when using GPU particles, which offload much of the particle calculation to the GPU, it is important to consider the value of calculating particles that the player is too far away to see or properly evaluate.

For example, consider the fire particle system. If you look at it up close, you can see the embers and sparks rising into smoke. But if you look at it from a distance of several hundred meters, those embers are so small that a monitor or screen cannot even reproduce them. So why calculate them?

This is where Levels of Detail (LOD) come into play. The LOD system [11] allows you to set your own distance ranges at which your particle system will automatically simplify. Each range represents a different LOD. Simplification comes in the form of reduced property values, disabled modules, or even disabled emitters. For example, in the bonfire example above, it would be ideal to completely disable the emitter that was adding sparks to the overall effect when the player was too far away to see them.

Your particle system can have as many LODs as you need, and you can manually control the ranges for each one.

Distributions are a set of data types for processing data in special ways, such as using a range for a value or interpolating a value along a curve. Whenever your particle system requires randomization or the ability to change some aspect of the particle over time, you will use a distribution to control that property.

Many properties found in Cascade modules can have different distributions applied to them. The actual value of the property is then set in the distribution.

9. Niagara

Why re-invent visual effects for Unreal Engine? Unreal Engine continues to expand its user base and is now used in many industries outside of game development.

Unreal Engine users are more diverse than ever, ranging from design students, to small indie developers, to large professional studio teams, to individuals and companies outside the gaming industry. As they move forward, Epic's developers will not know everything about every industry that uses the Unreal Engine. They wanted to create a visual effects (VFX) system that would work for all users across all industries.

They wanted to create a new system that would give all users the flexibility to create the effects they wanted. Their goals for the new VFX system were:

1. Put complete control in the hands of the artists.
2. The ability to program and customize on every axis.
3. Better tools for debugging, visualization, and productivity.
4. Support for data from other parts of the Unreal Engine or external sources.

Total user control begins with access to data. Epic Games wants the user to be able to use any data from any part of the Unreal Engine, as well as data from other applications. So they decided to give the user everything.

In order to make all this data available to a user, you need to determine how someone can use the data. Namespaces provide containers for hierarchical data. For example, Emitter.Age [12] contains data about an emitter; Particle.Position contains data about a particle. A parameter map is a particle payload that contains all the attributes of a particle. This makes everything optional.

Any kind of data can be added as a particle parameter. We can add complex structures, transform matrices, or boolean flags. We can add these or any other data types and use them to simulate effects.

There are advantages to both the stack paradigm (as used in Cascade) and the graph paradigm (as used in Blueprints). Stacks give users modular behavior and readability. Graphs give users more control over behavior. This new effect system combines the best of both paradigms.

Modules work in a graphical paradigm - we can create modules from HLSL in the script editor using a visual node graph. Modules interact with common data, encapsulate behavior, and are compiled together.

Emitters work in a stacked paradigm - they serve as containers for modules and can be stacked to create different effects. An emitter is single-purpose, but it is also reusable. Parameters are passed from modules to the emitter level, but you can change modules and parameters in the emitter.

Like emitters, systems work in a stacked paradigm and also use a sequencer timeline to control the behavior of emitters in the system. A System is a container for Emitters. The system combines these emitters into a single effect. When editing a system in the Niagara editor, we can change and override any parameter, module or emitter in the system.

The particle simulation in Niagara works conceptually like a stack - the simulation flows from the top of the stack to the bottom, executing the modules in order. Importantly, each module is assigned to a group that describes when the module is executed. For example, modules that initialize particles or act when a particle appears belong to the Particle Spawn group.

Within each group, there may be several stages that are invoked at certain points in the system's life cycle. Emitters, systems and particles have Spawn and Update stages by default. Spawn stages are invoked in the first frame in which the group exists. For example, systems invoke their spawn stage the first time the system is created and activated on a level. Particles invoke their spawn stage whenever the emitter emits a particle, and spawn instructions are executed for each new particle created. Update stages are invoked in every frame where a system, emitter or particle is active.

Рисунок Г.8 – Восьма сторінка статті в журналі «Біоніка інтелекту»

There are also advanced steps such as events and simulation steps that can be added to the spawn and update flow. Events are called whenever a particle generates a new event and the emitter is configured to handle that event. Where possible, event handler steps occur in the same frame, but after the original event. Simulation steps are an advanced GPU feature. This feature allows you to run multiple sleep and update stages in sequence and is useful for building complex structures such as fluid simulations.

By adding each module to a stage (update, spawn, event, or simulation) in a group (system, emitter, or particle), we can control when the module runs and what data it processes. Stack groups are associated with namespaces that define what data the modules in that group can read or write.

For example, modules executing in the System group can read and write parameters in the System namespace, but can only read from parameters that belong to the Engine or User namespaces. As execution moves down the stack from the System group to the Emitter group, modules executing in the Emitter group can read and write parameters in the Emitter namespace, but can only read from parameters in the System, Engine, and User namespaces. Modules in a particle group can only read from parameters in the System and Emitter namespaces.

Since modules in emitter groups can read parameters in the system namespace, a simulation relevant to all emitters can be performed once by modules in the system group, and the results of this simulation (stored in the system namespace) can be read by modules in the emitter group in each individual emitter. This continues with modules in the Particle group, which can read parameters in the System and Emitter namespaces.

In other respects, Niagara is very similar to Cascade. This system is more advanced, more flexible, and has endless possibilities for extending functionality, although this makes the process of creating particles more complicated. In this subsection, the main differences and architectural features between Cascade and Niagara have been presented, since the process of automated optimization in these systems is conceptually different.

10. Optimizing particle simulation systems

When creating a game, we can have a lot of variation in the FX workload depending on the composition of the scene. Sometimes it may be necessary to take steps to manage performance, such as dropping instances outside a certain range or instances that exceed a certain budget.

Effect Type resources allow you to configure a set of settings once and then apply them to a collection of Niagara systems.

The Effect Type object allows you to configure several different methods for selecting systems that exceed your budget. All of these options are available under the Budget Scaling heading.

Maximum global budget usage (see Figure 6): This option allows you to set a budget above which any system will be discarded. Typically, this setting is set to a value between 0 and 1, corresponding to a percentage between 0 and 100%. You can set it to 1.5 if you want the system to be more permissive. This means that once a system reaches this percentage of your budget, it will be discarded. This is the best option if you value performance over appearance.

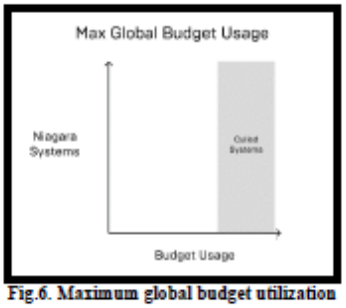

Fig.6. Maximum global budget utilization

Maximum Distance Scale by Global Budget Utilization: This option allows you to customize the curve to determine how the distance at which you select systems decreases as your budget usage increases. For example, if your budget is very high, Niagara will only render those that are nearby, not those that are far away.

Maximum Instance Scale by Global Budget Usage: This option allows you to configure a curve that determines how the number of instances in your tier will decrease as budget usage increases. This will scale down all instances of all systems that are subject to this type of effect.

Maximum scaling of system instances by global budget usage: This option allows you to customize the curve (see Figure 7) that determines how the number of instances at your tier decreases as budget usage increases. However, with this option, instead of dropping all instances on all systems, you drop a certain number of instances for each system.

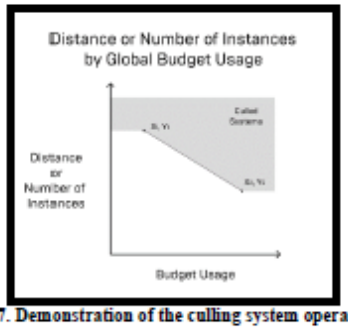

Fig.7. Demonstration of the culling system operation

For each of these three parameters, which take the values Start X, Start Y, End X, and End Y, these values define a linearly interpolated curve. Anything above this curve is discarded. For an example of what the curve will look like, see the diagram below.

Рисунок Г.9 – Дев’ята сторінка статті в журналі «Біоніка інтелекту»

In the grand scheme of things, particle count plays a very small role in performance. Regardless of whether the screen is split or not, material complexity and screen coverage (overdraw) are your clear enemies when it comes to the overall cost of a given system. A simple emissive spark, with nothing more than a texture multiplied by vertex colors and connected to an emissive input in an unlit material, follows only a handful of instructions. You can create them in droves all day long, and the overall impact on your productivity is likely to be very small. Sprites are small, which means screen coverage is low, and the complexity of the material makes them cheap and fast to render. The number of vertices isn't really something you need to worry about in the long run unless you're reaching really extreme numbers (hundreds, thousands, or more).

A much bigger impact on overall performance is the number of instructions for your materials. For materials like fire and smoke, there are basically two ways to go. The first is to create a more complex material for your effect. In the fire example, you would create fewer sprites and let the randomness and complexity of the advanced material do the work of bringing the emitter to life. Another option is to use a cheaper material and spawn more sprites, keeping the overall cost the same but allowing more particles to do your job in achieving randomness, as opposed to a more complex material. Keep in mind that material costs decrease exponentially with distance (a quadrilateral drawn on the screen twice as far away from the camera costs 4 times less due to the fact that the total pixel area decreases exponentially with distance, reducing the number of pixels that are dragged).

In our case, we need to analyze how expensive our materials are, how many sprites we create, and how close to the screen we will approach these effects. These three properties are the main decision makers in terms of the complexity of the emitter, and they all need to be balanced.

In general, focus on reducing the complexity of your materials as a way to improve performance, and always be aware of potential drag when you are working with emitters as a whole. Don't get hung up on particle counts unless you are generating extreme numbers of particles, or you are generating meshes using mesh emitters that have extreme numbers of vertices.

Conclusions

After analyzing the industry, we came to the conclusion that Unreal Engine is the most suitable for our tasks. Unity is more popular among developers for portable devices, but at the same time it has a less developed VFX creation system. The system implemented in Unity has not been globally updated since the creation of the engine, and the capabilities implemented in it are not flexible enough to conduct a pure experiment and analyze possible optimization strategies. At the same time, Unreal Engine has the most advanced VFX creation system - Niagara, which provides the best opportunity to study and optimize particle simulation systems, as it allows not only to use ready-made solutions, but also to write code independently.

The comparison of Niagara and Cascade showed a fundamental difference in the principles of optimizing particle simulation systems. Only one principle is relevant now, as it was before - reducing the number of particles, but as practice shows, this is often not enough. The change in the principles of optimization of these systems shows that the search is still ongoing and the decisions made earlier by the engine developers are no longer relevant. This suggests that new solutions may also be imperfect, as they try to solve the optimization problem universally, rather than in the best way for mobile devices.

For this reason, further analysis of existing optimization methods and their improvement or creation of new ones is a relevant and promising task. To this end, we plan to study the principles of VFX, existing optimization methods, and conduct tests on portable devices.

Conflict of Interest

The authors declare no conflict of interest.

References

[1] *Матюсов, Д.І., Лановий, О.Ф.* Методи спрощення опрацювання систем симуляцій неоднорідних часток у середовищі Unreal Engine 4 // Elektron. model. 2023, №45(2) с. 95-107 https://doi.org/10.15407/emodel.45.02.095 (дата доступу 15.12.2023)

[2] Cascade Particle Systems // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/ (дата доступу 15.12.2023)

[3] Niagara Visual Effects // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Niagara/ (дата доступу 15.12.2023)

[4] *Лановой А.Ф., Лановой А.А.* Моделирование поведения толпы на основе дискретно-событийного мультиагентного подхода // Східно-Європейський журнал передових технологій, 2014, №4(70), с. 52-57

[5] GPUSprites Type Data // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/Reference/TypeData/GPUSprites/ (дата доступу 15.12.2023)

[6] Collision Modules // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/Reference/Modules/Collision/ (дата доступу 15.12.2023)

[7] Event Modules // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/Reference/Modules/Event/ (дата доступу 15.12.2023)

[8] VFX Optimization Guide // Unreal Engine Documentation. Режим доступу: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/Optimization/ (дата доступу 15.12.2023)

[9] *Матюсов, Д.І., Лановий, О.Ф.* Проблеми оптимізації графіки під пристрої віртуальної реальності // ΛΟΓΟΣ.ONLINE, 2020, №14. http://soi.citefactor.org/10.11232/2663-4139.14.04 (дата доступу 15.12.2023)

[10] Особливості підготовки 3D моделей для використання у VR проектах // Science, Research, Development. Режим доступу: http://www.xnsh.com.ua/files/118_01_xi_2021.pdf#page=35 (дата доступу 15.12.2023)

[11] Порівняння методів текстурування моделей для мобільних платформ // Science, Research, Development. Режим доступу: http://www.xnsh.com.ua/files/118_01_xi_2021.pdf#page=37 (дата доступу 15.12.2023)

[12] Дослідження інструментів та засобів оптимізації 3D-графіки в комп'ютерних іграх та їх застосування до ігор у жанрі "First-person Shooter" // Електронний архів Харківського національного університету радіоелектроніки. Режим доступу: https://openarchive.nure.ua/server/api/core/bitstreams/e8582e45-10b9-44bf-aabc-cb0b120389ee/content (дата доступу 15.12.2023)

Рисунок Г.10 – Десята сторінка статті в журналі «Біоніка інтелекту»

ДОДАТОК Д

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-22-4 (група)

Шульга Максим Віталійович
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
	7.3 Нумерація сторінок звіту	
	7.4 Нумерація розділів, підрозділів, пунктів, підпунктів	
	7.5 Рисунки	
	7.6 Таблиці	
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні познаки	
	7.15 Додатки	

зауважень немає

Експерт _____ (підпис) Олена ОЛІЙНИК (прізвище, ініціали)

09.06.2024