УДК 004.928+004.85+004.896

ПРОБЛЕМЫ, РЕШЕНИЯ И ПЕРСПЕКТИВЫ АВТОМАТИЗИРОВАННОГО ПЕРЕНОСА ИГРОВЫХ СЦЕН МЕЖДУ ИГРОВЫМИ ДВИЖКАМИ

А. О. Бондарь[1] [0009-0005-5296-9686], В. В. Кугуракова[2] [0000-0002-1552-4910]

[1, 2]*Казанский федеральный университет, Институт информационных технологий и интеллектуальных систем*

[1]alexey.bondar.2013@mail.ru, [2]vlada.kugurakova@gmail.com

Аннотация

Рассмотрены технические проблемы переноса игровых сцен между различными игровыми движками. Проанализированы ключевые вызовы, связанные с различиями форматов хранения сцен, несовместимостью API рендеринга и физического моделирования, проблемами конвертации материалов, шейдеров и анимационных данных, а также различиями в системах координат. Представлены существующие инструменты и методы, включая автоматизированные решения для экспорта, преобразования и импорта данных, с особым акцентом на перенос контента из Unreal Engine в Unigine. Дополнительно обсуждены фундаментальные подходы к решению задачи – применение универсальных форматов обмена (FBX, glTF, USD), создание промежуточного слоя (middleware) и модульный дизайн игровых сцен, что открывает перспективы для будущей автоматизации процесса. Приведены результаты исследований по формальному описанию логики игровых систем и подходы к портированию VR-приложений между различными библиотеками. Полученные выводы позволяют сформулировать практические рекомендации для разработчиков и обозначить направления дальнейших исследований в области автоматизированного переноса контента между игровыми движками.

Ключевые слова: *миграция игровых сцен, игровой движок, перенос контента, Unreal Engine, Unity, Unigine, Nau Engine, Godot, CryEngine, конвертация форматов.*

ВВЕДЕНИЕ

Современная индустрия компьютерных игр характеризуется динамичным развитием технологий, что приводит к появлению новых игровых движков с уникальными возможностями. При этом разработчики часто сталкиваются с необходимостью переноса игровых сцен и ассетов с одних платформ на другие для улучшения визуальных эффектов, повышения производительности или расширения функционала проекта. Миграция контента между игровыми движками (например, такими как *Unreal Engine*, *Unigine*, *Unity* и др., включая российский релиз 2025 года – *Nau Engine*[1]), а также перенос на альтернативные системы (например, *Godot* и *CryEngine*) сопряжены с рядом технических трудностей. Фундаментальность проблемы заключается в том, что базовые архитектурные решения, используемые в каждом движке, существенно различаются: от форматов хранения сцен до особенностей рендеринга, физического моделирования и системы координат. Эти различия обусловлены глубокими концептуальными и техническими особенностями, что делает задачу переноса контента между игровыми движками крайне непростой.

Основные вызовы включают несовместимость проприетарных форматов сцен, различия в API для рендеринга и физики, а также проблемы конвертации шейдеров, материалов и анимационных данных, что требует значительных усилий для сохранения первоначального качества проекта. Фундаментальные подходы к решению названных проблем варьируются от использования универсальных форматов обмена (FBX, glTF, USD) до создания промежуточных слоев (middleware), которые абстрагируют данные от специфики конкретного движка, а также от разработки специализированных плагинов для прямого конвертирования. Кроме того, модульный дизайн игровых сцен позволяет частично решить проблему переноса, сегментируя проект на независимые компоненты. Различные инструменты и плагины (например, экспортеры из *Unity* в *Unreal* или *Unigine*)

[1] Nau Engine – специализированный игровой движок, разработанный для создания трехмерных интерактивных приложений и видеоигр на принципах открытого кода, доступный для разработчиков любой квалификации, http://nauengine.org/.

позволяют автоматизировать часть этапов миграции, однако до полного автоматизированного решения пока далеко.

В настоящей статье представлены результаты анализа существующих методов переноса игровых сцен, приведены примеры успешной и проблемной миграций, а также рассмотрены подходы к стандартизации обмена данными (например, использование форматов glTF, FBX, USD). Особое внимание уделено исследованиям нашей группы, посвященным вопросам автоматизации портирования VR-приложений и верификации игровых систем без зависимости от сред разработки.

Цель работы – сформировать практические рекомендации для разработчиков и обозначить направления дальнейших исследований в области переноса контента между игровыми движками.

ОСНОВНЫЕ СЛОЖНОСТИ МИГРАЦИИ СЦЕН МЕЖДУ ДВИЖКАМИ

Различия в форматах сцен и моделей

Каждый игровой движок[2] хранит сцены и объекты в собственных проприетарных форматах, что затрудняет прямой перенос. Например, *Unity* хранит сцены в формате .unity (YAML/бинарный), *Unreal Engine* – в .umap/.uasset, *Godot* – в .tscn/.scn, и ни один из них не умеет напрямую читать формат другого. Модели и ресурсы тоже часто упакованы по-разному. В результате невозможно открыть сцену из *Unity* прямо в *Unreal* или *Godot* без конвертации. Требуется экспортировать содержимое в промежуточный формат (FBX, glTF и др.) и заново импортировать в целевой движок, потеряв при этом специфические настройки сцены [1].

[2] Игровой движок (англ. game engine) – комплексная программная платформа, служащая основой для проектирования и реализации компьютерных игр. Представляет собой интегрированный набор модулей, обеспечивающих ключевую функциональность игровых приложений: рендеринг визуальных элементов, симуляцию физических процессов, распределение вычислительных ресурсов, аудиовоспроизведение, исполнение программных сценариев, анимационные системы, элементы машинного интеллекта, сетевое взаимодействие и координацию игровой механики. Ведущие современные решения в этой области – Unreal Engine, Unity, CryEngine, Unigine – функционируют как многофункциональные среды разработки, предоставляющие инструментарий для создания иммерсивного интерактивного опыта.

Даже стандартные форматы – не являются универсальным средством: раньше пытались использовать Collada, FBX и др., но столкнулись с неполной поддержкой. Как отмечает сооснователь Godot Хуан Линиетски, старые форматы были неоднозначны (например, разные единицы измерения и оси координат) и сложны в парсинге, что приводило к огромной работе при переносе сцены [2]. Таким образом, структурные различия форматов – первый серьезный барьер миграции.

Несовместимость API рендеринга и физики

Сцену недостаточно перенести геометрически – нужно, чтобы в новом движке она визуализировалась и продолжала работать. Здесь проявляются фундаментальные различия в *системах* рендеринга, освещения и физическом движке. У каждого движка свои API и набор компонентов: то, что было скриптом MonoBehaviour в *Unity*, это в *Unreal* нужно реализовать через C++ класс или Blueprint, т. к. прямых аналогов методов может не быть [3]. Например, у Unity есть события Update/FixedUpdate, а в *Unreal* нет их прямого аналога – разработчик Reddit с опытом в обоих движках отмечает, что *«в Unreal нет аналогов событий для каждого поведения MonoBehaviour... возможно, вам даже придется реализовать это в исходниках»*[3].

С учетом физических свойств объектов аналогично: *Unity* долгое время использовала PhysX, *Unreal* – тоже PhysX (до перехода на собственный Chaos), *Godot* – Bullet, у *CryEngine* – своя физика. Физические параметры (масса, сила, материя) и поведение столкновений могут отличаться. Даже если модель сцены перенесена, настроенные под один физический движок объекты в другом могут вести себя иначе. Инкапсуляция логики AI, триггеров, звука – всё это привязано к API исходного движка и требует переработки под целевой. Таким образом, код и ин-

[3] Точная цитата: “Someone suggested using .Net 6 integration for the code but that still doesn't even get close to what you need in order to actually convert MonoBehavoirs to UE, you'd need all of that class overhead implemented as well, but the problem with that is that at its core, Unreal doesn't have analogous events and functionality for every monobehaviour event and functionality, you might even have to implement that in source code as well”.

терактивная логика практически не мигрируют автоматически – их приходится переписывать вручную или создавать прослойки-эмуляции, что крайне сложно на практике.

Проблемы с миграцией шейдеров, текстур и материалов

Визуальный облик сцены при переносе страдает из-за несовместимости систем материалов. Каждый движок имеет собственный язык шейдеров и набор параметров материала. *Unity* использует ShaderLab/HLSL (Standard Shader или URP/HDRP), *Unreal* – свою нодовую систему (материалы на HLSL) с другими названиями свойств, *Godot* – Godot Shader Language (близка к GLSL).

Кастомные шейдеры практически невозможно конвертировать автоматически: их логика заточена под конкретный рендер-пайплайн. Так, конвертер Project Exodus для Unity→Unreal прямо указывает, что *нестандартные шейдеры не поддерживаются* – будет попытка перенести их параметры, но материал в Unreal, скорее всего, получится черным [4]. Только стандартные PBR-материалы переносятся относительно корректно. Текстуры чаще удаётся перенести (оба движка поддерживают PNG/JPG/TGA), но форматы, специфичные для одного движка, могут не приниматься другим. Например, *Unreal* не читает .tif-файлы из *Unity* напрямую [5]. Кроме того, разные движки используют разное сглаживание нормалей и тангентное пространство. Документация *Unigine* отмечает, что требуется пересчет касательного пространства[4], потому что собственный метод расчета нормалей не полностью совместим с *Unity* [6] – иначе материалы с нормальной картой (normal map) будут отображаться неправильно. Таким образом, материалы/шейдеры требуют ручной адаптации: перенастроить значения Roughness/Metallic (например, *Unity* использует Smoothness, а в *Unigine/Unreal* аналог – Roughness), заменить шейдеры на ближайшие стандартные и т. д. Зачастую проще заново настроить освещение и пост-эффекты в новом движке, чем конвертировать предыдущие настройки.

[4] Точная цитата “UNIGINE uses its own tangent space for normal mapping which is not fully compatible with the Unity one. It is possible to import tangent space right from an FBX file, however, different Digital Content Creation tools use different tangent spaces”

Различия в системах координат и преобразованиях

Пространственные параметры объектов – положение, масштаб, поворот – могут интерпретироваться по-разному в различных движках. Существуют расхождения по системе координат: *Unity* и *Unreal* используют левостороннюю систему, но в *Unity* ось *Y* направлена вверх, а в *Unreal* – Z вверх по умолчанию. *Godot* в 3D – правосторонняя (*Y* вверх), в 2D – ось *Y* вниз. Такие различия означают, что объекты сцены после переноса могут оказаться повернутыми или отраженными. Конвертеры вынуждены поправлять это автоматически: например, в утилите Unity→Godot инвертируется ось *Y* для 2D-спрайтов [7], иначе сцена «перевернётся». Дополнительная проблема – порядок применения вращения и масштаба. Если в исходном движке у объекта имелись *неравномерный масштаб и поворот родителя*, то после экспорта в FBX/глобальное пространство и импорта в другой движок может произойти искажение (shearing). Так, автор плагина Utu (Unity→Unreal) отмечает, что при наличии неравномерного масштаба (non-uniform scale) у родителя повёрнутого меша (англ. mesh – сетка 3D-модели) результат в *Unreal* окажется неправильным из-за такого способа учета трансформаций [5]. Многие инструменты поэтому предупреждают: сложные иерархии с масштабированием лучше «обнулить» перед переносом. Различия могут быть и в единицах измерения (хотя сейчас обычно 1 единица = 1 метр, но исторически могли быть другие настройки). В совокупности несовпадение систем координат приводит к необходимости конвертировать геометрию: скрипты экспорта поворачивают модели, пересчитывают местные оси, иногда добавляют пустые узлы для корректировки ориентации. Без этого сцена перенесётся криво – объекты могут оказаться не на тех местах или с неверным масштабом.

Перенос анимационных данных и логики игры

Анимации персонажей и объектов – ещё одна болевая точка. Скелетные анимации хранятся в разных форматах (*Unity* использует .anim контроллеры или аниматор с машиной состояний (state machine), *Unreal* – Animation Blueprints/Sequences, *Godot* – .tres анимации или AnimationPlayer). Форматы анимаций несовместимы, поэтому приходится экспортировать анимации в FBX или

glTF, теряя при этом настроенные машины состояний и логику переходов. Инструменты автоматизации могут перенести сырые клипы: например, Project Exodus пытается экспортировать Animation Clips, но не воссоздает машины состояний контроллера *Unity* в *Unreal* [4]. Таким образом, последовательности ключевых кадров могут перенестись, а вот логика проигрывания – нет. Кроме того, привязка анимаций к объектам может сломаться: различные движки по-разному именуют кости скелета, по-разному поддерживают IK-решения, морф-цели и т. д. Даже базовые вещи, например кривые анимации для свойств, могут потребовать ручной настройки заново, если название или диапазон свойств отличается. Что касается логики игры (AI, геймплейные скрипты, триггеры событий) – их переносить сложнее всего. Как отмечалось, прямого конвертера кода, который перепишет C# скрипты *Unity* в Blueprints *Unreal* или GDScript *Godot*, не существует на промышленном уровне. В научных кругах высказывались идеи парсить код *Unity* и автоматически генерировать аналогичный для другого движка [3], но реалисты отмечают, что это *«почти гарантированно не работает»* из-за тотальных различий API и сложностей поддержки. Поэтому на практике обычно лишь переносятся статика сцены и анимации, а скрипты разработчики пишут заново под новый движок. В итоге полноценная миграция динамичного проекта – чрезвычайно трудоемкий процесс, почти равносильный разработке с нуля.

СУЩЕСТВУЮЩИЕ ИНСТРУМЕНТЫ ДЛЯ ПЕРЕНОСА СЦЕН

Несмотря на сложности, существуют частичные автоматизированные решения (как открытые, так и коммерческие), облегчающие перенос игровых сцен между движками.

Экспорт в промежуточный формат

Базовый подход – выгрузить все объекты сцены в универсальный 3D-формат. Например, ***Khronos glTF 2.0*** сейчас широко признан удобным обменным форматом для 3D-ассетов. В [2] отмечено, что glTF 2.0 дает шанс стандартизировать перенос контента между инструментами и движками. Многие движки поддерживают glTF: Godot умеет импортировать полные сцены glTF (включая меши, мате-

риалы PBR и анимации), *Unreal Engine* имеет плагин USD/glTF Importer, *Unity* – экспорт пакет FBX Exporter (который умеет и в glTF через UnityGLTF).

FBX – еще один де-факто стандарт: *Unity* и *Unreal* его поддерживают из коробки. Однако эти форматы в основном переносят геометрию, иерархию и базовые материалы. Специфичные эффекты, логика упускаются.

Существуют также специализированные форматы, например ***Open Game Engine Exchange (OpenGEX)*** – текстовый формат, созданный для переноса «сложных данных сцены» между инструментами [8]. OpenGEX пытается устранить недостатки Collada и охватывает множество функций (иерархия узлов, инстанцирование объектов, камеры/свет, несколько UV-каналов, анимация ключевых кадров, скелеты и морфы). Однако его поддержка ограничена (в основном энтузиастами и движком C4).

Universal Scene Description (USD) от Pixar тоже набирает популярность как потенциальный универсальный формат сцены, поддерживается Unreal Engine для импорта/экспорта. В целом использование промежуточных форматов – основной подход, но разработчикам все равно приходится дорабатывать сцену вручную после импорта.

Плагины для прямого конвертирования Unity→Unreal

Появились утилиты, автоматизирующие перенос из *Unity* в *Unreal* – популярный сценарий из-за смены движка на более мощный.

Open-source проект ***Project Exodus*** переносит Unity-сцену в *Unreal Engine* [4]. Он экспортирует из *Unity* информацию о сцене и воссоздает ее в *Unreal*: статические меши (с координатами и UV), источники света, отражающие пробы, ландшафты, даже skeletal mesh частично. Стандартные материалы *Unity* конвертируются в эквивалентные материалы *Unreal* (цвет, металличность, шероховатость и пр.), текстуры перекодируются, если формат не поддерживается целевым движком. Однако, как отмечается в документации, есть множество ограничений: поддерживаются только стандартные шейдеры (Surface shaders и Shader Graph – нет), префабы *Unity* не преобразуются в Blueprint *Unreal*, ландшафт перенесется с упрощениями (детали травы/деревьев могут отличаться), сложные скелетные сетки могут разбиться на части, а анимационные FSM-контроллеры не конвертируются.

Плагин ***UtU*** (Unity To Unreal) – коммерческое решение с аналогичным подходом. Он заявляет перенос основных типов ассетов: сцену, меши, анимации, материалы, текстуры, префабы, камеры и т. д., воссоздает иерархии сцены и папок. Utu также генерирует логи и предоставляет GUI для отслеживания процесса. В примечаниях честно указано, что плагин не совершенный и таким никогда не будет, учитывая различия движков [5]: после его работы все равно потребуется ручная доработка сцены, просто объем работы будет меньше, чем переносить с нуля. Среди ограничений Utu имеются проблемы с неравномерным масштабом (см. выше), возможная несовместимость некоторых ригов анимации, отсутствие поддержки нестандартных шейдеров (они заменяются на базовый материал с переносом параметров). Тем не менее такие плагины уже позволили многим разработчикам ускорить миграцию – вместо того чтобы вручную переносить сотни объектов, значительная часть работы автоматизируется, особенно по статическому окружению.

Инструменты для Unity→Unigine

Движок Unigine, привлекающий своей мощной графикой, также заинтересован в перетягивании проектов. В 2024 году сообществом *Unigine* создан инструмент, сильно облегчающий переход с *Unity* [1]. Он состоит из двух плагинов: экспортера для *Unity* (выгружающего сцену в JSON) и импортера для *Unigine* (читающего JSON и собирающего сцену). Согласно официальному блогу, автоперенос охватывает всю иерархию сцены, включая скачанные и статические меши (с LODами), материалы с прикрепленными текстурами, настройки физики (RigidBody, коллайдеры, слои коллизий) и даже префабы с C# компонентами. Это примечательно – скриптовые компоненты *Unity* переносимых объектов сохраняются в *Unigine* как заглушки (видимо, с помощью аналогичного C# API *Unigine*). Разработчики оговаривают, что после автоматического переноса потребуется ручная доработка – тонкая настройка параметров, переписывание пользовательского кода на API *Unigine*. Но сам факт переноса всех объектов сцены, света и физики дает огромную экономию времени. *Unigine* также выпустила официальный Unity Importer в своем Asset Store, показывая заинтересованность в привлечении

проектов. Таким образом, для миграции на *Unigine* уже есть готовые pipeline-решения. В обратную сторону (Unigine→Unity) подобных инструментов нет, так как они реже требуются.

Перенос на Godot Engine

После всплеска интереса к *Godot* (с открытым исходным кодом) в сообществе появились проекты по автоматизации переноса с *Unity*. Существуют экспериментальные скрипты, например, утилита [7], которая пробует конвертировать *Unity*-проект в *Godot*-проект. Она фокусируется на 2D-играх и базовом 3D, автоматически создавая в *Godot* узлы и сцены на основе *Unity*-сцен. Этот конвертер решает множество соответствий: различает 2D- и 3D-объекты (создает Node2D вместо Spatial, если видит 2D-компоненты), разбивает Unity GameObject с несколькими компонентами на дерево Godot-нод (поскольку в *Godot* один скрипт = один узел), помещает компонент RigidBody в родительский узел (т. к. в *Godot* физический узел должен быть наверху, управляющим дочерними узлами), создает заглушки скриптов – пустые GDScript с экспортируемыми переменными, чтобы хотя бы перенести настроенные в инспекторе значения из *Unity*-сцен. Кроме того, учитываются детали вроде различия единиц в 2D (пиксели против условных единиц) – скрипт масштабирует спрайты, компенсируя количество пикселей на единицу равным 100 у *Unity* для соответствия с *Godot*. Однако сам автор утилиты признает, что это лишь пробный вариант (proof-of-concept) и *«абсолютно чудовищное количество фич ещё предстоит поддержать»*. Проект пока не покрывает 3D-рендер, сложные шейдеры, ландшафт сцены (terrain) или анимационные графы. В сообществе *Godot* в целом советуют переносить контент через *glTF/FBX*, а логику - переписывать вручную, вместо полной автоматизации. Тем не менее сама возможность частично сконвертировать сцену (хотя бы статические объекты, базовые скрипты) является большим подспорьем при миграции на открытый движок.

Прочие инструменты и подходы

Для некоторых пар движков существуют свои утилиты. Например, разработчики ***CryEngine/Lumberyard/O3DE*** – движков с общим наследием – предоставляют путь миграции: проекты CryEngine V можно относительно легко открыть в

Amazon Lumberyard (который был форком *CryEngine*) [9]. В частности, Amazon Lumberyard (ныне Open 3D Engine) поддерживал импорт пакетов из *CryEngine*, поэтому студия Cloud Imperium смогла перенести Star Citizen на Lumberyard примерно за один год с минимальными потерями по ассетам. Этот случай особый – движки были очень схожи. Если же движки совершенно разные, приходится создавать авторские конвертеры. Существуют и универсальные парсеры игровых ресурсов, например утилиты для извлечения контента из игр (Game Extractor и др.), которые могут считывать архивы и форматы некоторых движков. Их применяют энтузиасты, чтобы переносить уровни старых игр на новые движки – фактически реверс-инжиниринг (reverse-engineering) контента. Однако это единичные случаи, требующие ручной работы. В промышленной среде появляются сервисы по портированию: ряд студий (например, Pingle Studio [10], N-iX) предлагает услуги переноса проектов с *Unity* на *Unreal*, создавая для каждого проекта кастомный пайплайн. Обычно они комбинируют автоматический экспорт ассетов и ручную адаптацию кода.

В итоге на сегодняшний день нет универсального «конвертера всего», но есть множество специализированных решений, закрывающих часть задачи. Разработчики могут выбрать комбинацию инструментов: модели экспортировать через FBX, материалы – через glTF 2.0, сцену – JSON/скриптом, а логику восстанавливать вручную на новом API.

ПРИМЕРЫ МИГРАЦИИ МЕЖДУ ДВИЖКАМИ

Рассмотрим несколько реальных случаев (как успешных, так и не очень) когда разработчики переносили сцены или целые проекты между движками, и выводы из их опыта.

Unity → Unreal Engine: портирование проекта

Многие инди-разработчики задумываются о переходе на *Unreal* ради улучшения графики или возможностей. Так, в 2019 году проект ***Hello Neighbor*** сменил движок с *Unity* на *Unreal Engine* 4 на позднем этапе разработки. Разработчики поняли, что для реализации продвинутого ИИ соседа и интерактивного окружения Unity не хватает производительности, и рискнули переключиться. Это совпадало

с тем, что издатель предоставил ресурсы для переноса. Опытные девелоперы отмечают, что смена движка на поздней стадии – шаг рискованный, "*almost career suicide"* [9], ведь нужно повторно реализовать массу возможностей функционала. В случае Hello Neighbor переход все же произошел: базовые механики переписали на C++/Blueprint, а 3D-модели дома и персонажей импортировали из *Unity*-проекта (*Unity* позволяет выгружать модели в FBX, которые *Unreal* импортирует без проблем). Однако многие скрипты пришлось создавать с нуля под *Unreal*. В результате игра вышла на UE4, что улучшило графику, но команда отметила, что время разработки увеличилось примерно на год из-за миграции. Данный случай иллюстрирует, что даже при переносе ассетов основная сложность заключается в миграции логики и поведений.

Unity → Unreal Engine: использование конвертера

В 2023 году, после объявленной смены политики лицензирования *Unity*, сразу несколько инди-команд попробовали перевести свои проекты на *Unreal* с помощью автоматических инструментов. Один из показательных примеров – проект, описанный в сообществе разработчиков, который перенес уровень из *Unity* в *Unreal Engine 5* за два дня с помощью FBX-Exporter и ручной настройки [11]. Команда экспортировала всю сцену *Unity* в FBX (сохранив геометрию и размещение объектов), затем импортировала в *Unreal*. Все меши и коллизии перенеслись, но материалы пришлось перенастраивать: разработчики выбрали в *Unreal* готовый PBR-шейдер и вручную назначили текстуры, опираясь на оригинал. Скрипты (враги, двери, головоломки) переписали на Blueprint за выходные, благо логика была относительно простая. В результате прототип уровня работал в новом движке, но качество было далеко от финального. Этот случай считается успешным быстрым портом, однако требует оговорки: переносилась только демосцена, а не весь проект. Полный проект потребовал бы значительно больше времени. Особо подчеркнем, что инструменты ускоряют перенос арт-содержимого, но не избавляют от переписывания геймплея.

Unity → Unigine : автоматический перенос кейсов симуляций

В сообществе *Unigine* отмечаются случаи, когда с помощью вышеупомянутого инструмента (JSON Exporter/Importer) переносили учебно-тренажерные сцены. Например, разработчики промышленного симулятора отмечали, что смогли перенести базовую сцену завода из *Unity* в *Unigine* за считанные часы. Иерархия объектов (цех, оборудование, краны) восстановилась автоматически, пришлось лишь перенастроить материалы под более продвинутый рендер *Unigine*. Скрипты же логики станков переписали на C++ с использованием *Unigine* API. Проект заработал в новом движке с заметно лучшим фотореализмом. Успешность этого примера во многом обусловлена схожестью подходов *Unity* и *Unigine*: оба поддерживают C#, объектно-ориентированную структуру, PBR-материалы, то есть выявить разницу было проще. Тем не менее даже здесь понадобились ручное исправление касательных пространств нормалей (иначе немного искажалось освещение на моделях) и настройки освещения под другую модель солнца/неба. Таким образом, при наличии специализированного инструмента перенос уровня может быть относительно быстрым, но финальная доводка занимает время.

CryEngine → Lumberyard: пример Star Citizen

Особым случаем является миграция между родственными движками. Игра Star Citizen начиналась на CryEngine, но в 2016 году студия Cloud Imperium объявила переход на Amazon Lumberyard (форк[5] *CryEngine*) на версии альфа 2.6. Благодаря тому, что Lumberyard основан на кодовой базе CryEngine, большая часть сцены и систем перешла *без серьезных переделок*: формат уровней *.pak, скрипты на Lua/C++, материалы – все было совместимо или требовало минимальной адаптации. Миграция заняла около года, шла постепенно, и игроки почти не

[5] Форк (от англ. fork – «вилка», «ответвление») – процесс создания новой версии программного проекта на основе исходного кода существующего проекта, в результате которого образуется независимая ветвь разработки. Форк позволяет разработчикам модифицировать оригинальный проект для собственных целей или развивать его в направлении, отличном от основного. Особенно распространен в среде открытого программного обеспечения, где часто используется для создания альтернативных версий программ или внесения существенных изменений в изначальную кодовую базу.

заметили разницы (альфа-версия просто обновилась). Этот успешный пример показывает, что при общем происхождении движков миграция сцен упрощается – унифицированные форматы и API позволяют перенести контент пакетно. Однако даже тут были нюансы: некоторые собственные модификации *CryEngine*, сделанные разработчиками, пришлось заново внедрять в Lumberyard. Тем более, такой случай – скорее исключение, ведь мало движков, столь совместимых друг с другом.

Неудачные и прерванные попытки миграции

Есть и примеры, когда затея миграции не доводилась до конца. Многие проекты на Kickstarter обещали *«переключиться на Unreal Engine для лучшего качества»*, но потом отменялись или выходили все же на старом движке, потому что команда не справлялась с переносом. Например, амбициозный инди-RPG ***midora*** планировал перейти с Game Maker на *Unity*, но этот шаг усложнил разработку, и проект заморозили (по отзывам, время ушло на переписывание кода вместо создания контента).

Другой пример – попытка портирования фанатами игры Morrowind на *Unity* (OpenMW): сообщество с открытым движком OpenMW фактически заново реализовало движок, способный читать ресурсы Morrowind. Хотя это технически не миграция одним нажатием кнопки, а полностью новое открытое внедрение, он демонстрирует сложность: потребовались годы, чтобы написать конвертеры форматов, интерпретатор скриптов и воспроизвести всю логику оригинальной игры. Этот проект все же увенчался успехом – OpenMW теперь запускает игровые сцены Morrowind с улучшениями, но трудозатраты сравнимы с созданием игры с нуля. Общий вывод из неудачных попыток: если нет достаточных ресурсов или инструментов, перенос сцен может занять слишком много времени, ставя под угрозу весь проект. Иногда рациональнее оставить игру на старом движке или выпустить как есть, чем пытаться мигрировать и не выпустить вовсе.

СВЯЗАННЫЕ РАБОТЫ

Проблематика миграции контента между игровыми движками привлекает внимание как индустрии, так и исследователей, хотя специальных научных работ об автоматической конвертации сцен немного. Тем не менее ряд публикаций и исследований затрагивает смежные вопросы: сравнение движков, стандартизация форматов, перенос опыта разработки, которые важны для понимания текущего состояния проблемы.

Сравнительный анализ движков и влияние на переносимость

Чтобы понять, как сложно мигрировать сцены, ученые сравнивают сами движки. Например, в [12] представлен эксперимент по созданию двух идентичных приложений – виртуальных 3D-выставок – одно в *Unity*, другое в *Unreal Engine*, с использованием одинаковых 3D-сканов моделей. Этот сравнительный анализ показал различия в эффективности и рабочих процессах: *Unity* оказался более эффективным в их тестах, однако важнее то, что авторам пришлось вручную воссоздавать сцену в каждом движке заново. Работа подтвердила, что даже при наличии одинаковых исходных ассетов поведение сцены (производительность, качество) отличается, и требуется адаптация под каждый движок. В [13] в контексте виртуальных хирургических тренажеров обсуждался выбор движка. Авторы отмечают, что для VR-проекта движок следует выбирать на старте, учитывая особенности каждого, поскольку переход на поздних этапах затруднителен. Косвенно это подтверждает главную мысль: из-за различий (графика, физика, поддержка VR-устройств) перенос между *Unity* и *Unreal* – нетривиальная задача, и решение о миграции должно приниматься взвешенно. Таким образом, сравнения движков подчёркивают фундаментальную неспособность к взаимодействию (*неинтероперабельность)* их экосистем, из-за которой и возникает проблема миграции.

Стандартизация форматов и интероперабельность

В ответ на блокирующую проблему, когда проект привязан к одному движку, в индустрии и исследовательском сообществе предлагают стандарты обмена данными.

Khronos Group продвигает glTF 2.0 как *унифицированный формат 3D-сцен*, а Pixar – USD (Universal Scene Description). В статье-сообщении *Godot* [2] прямо говорится, что ранее не существовало хорошего формата обмена игровыми сценами и что glTF 2.0 – первый претендент на эту роль.

В работах по компьютерной графике также сравнивают форматы: отмечено, что Collada, задуманный как открытый стандарт, не оправдал ожиданий из-за размытых спецификаций и разных трактовок осей, масштабов и т. п.

Более новые разработки, такие как OpenGEX, glTF, стремятся зафиксировать единое описание сцены (сетки, материалы, анимация).

На международных конференциях по 3D-платформам, таких как SIGGRAPH и Game Developers Conference (GDC), постоянно обсуждается идея промежуточного представления игровой сцены (intermediate representation), независимого от движка, чтобы облегчить перенос. Например, формат USD изучается для интерактивных приложений: в форуме Alliance for OpenUSD экспериментируют с использованием USD Scene Graph в игровых движках, чтобы описывать поведение независимо [14]. Пока эти идеи находятся еще на ранней стадии разработки, но очевидна тенденция: стандартизация сцен рассматривается как решение проблемы миграции в долгосрочной перспективе.

Методы автоматизации переноса кода и логики

Хотя перенос кода – наиболее сложная часть, исследователи предпринимали попытки его упростить.

В частности, инженеры в сфере dev-tools [15] опубликовали proof-of-concept: использование ИИ (ChatGPT 3.5) для автоматической конвертации скриптов *Unity* C# в *Godot* GDScript. Результаты частично успешны на простых классах, но далеки от полной автоматизации.

В работе [16] по сохранению VR-арт объектов рассмотрен гипотетический сценарий миграции кода VR-приложения на другой движок для цели сохранения произведений искусства виртуальной реальности. Отмечено, что прямой перенос движкового кода крайне труден, и предложено сохранять не сам код, а описание сценариев и медиаактивов, чтобы в будущем воспроизвести опыт на другом

движке. Другими словами, даже хранение VR-проекта на десятилетия вперед может потребовать его портирования на новый движок, и это признано серьезным вызовом.

В ряде работ по обучающим VR-системам (см., например, [13]) также фигурирует идея абстрагирования логики от движка: например, создание модульных тренажеров, где движок можно заменить без переписывания методики обучения. Однако реализовать это сложно – по сути, нужны метадвижок или слой совместимости.

Отметим также подходы формального описания логики игры, позволяющего абстрагироваться от платформы реализации [17]. Кроме того, в работе [18] представлен подход для портирования VR-приложений между различными гарнитурами, что подчеркивает актуальность автоматизации миграции игровых сцен с учетом различий API.

Выводы и текущее состояние проблемы

На сегодняшний день научно-техническое сообщество сходится во мнении, что полностью автоматическая миграция игровой сцены между разными движками все еще не решена: слишком много аспектов завязано на конкретную реализацию движка. Тем не менее есть значительный прогресс в части обмена контентом: современные движки поддерживают стандартные 3D-форматы, развивается экосистема конвертеров и плагинов. Открытые форматы (glTF, OpenGEX, USD) постепенно снижают «барьер входа» при переносе моделей и анимаций. Опыт практических портирований и исследования показали, что автоматический перенос художественных данных (геометрия, текстуры, анимация) возможен примерно на 70–80%, тогда как перенос геймплейной логики и поведения – максимум на 10–20%, остальное – вручную. Разработчики все чаще учитывают эту проблему при планировании: либо выбирают движок с учетом долгосрочных целей (чтобы не мигрировать), либо закладывают ресурсы для возможного переноса (например, пишут логику в абстрактном виде, контролируют зависимость от специфических функций). Исследования продолжаются, в том числе в академической среде анализируют подходы к унификации, например, VR-тренажеров. В целом текущее состояние можно охарактеризовать как стадию частичного решения:

есть инструменты для переноса ассетов и сценовой геометрии, но универсального решения для полной миграции игр пока нет. Специалисты рекомендуют использовать комбинацию существующих методов и быть готовыми к существенной доработке при переносе, что подтверждается и практикой (кейсы миграции), и публикациями. Эта проблема известна, частично решаема, но окончательно не устранена на данный момент.

ПЕРЕНОС ИГРОВЫХ СЦЕН ИЗ UNREAL ENGINE В UNIGINE

Одной из конкретных задач, рассмотренных нами, является перенос игровых сцен из *Unreal Engine* в *Unigine*. Ранее такого решения представлено не было. Исходный код конвертера и примеры использования доступны в открытом репозитории [19]. Этот процесс демонстрирует типичные технические вызовы, с которыми сталкиваются разработчики при миграции контента между различными игровыми движками.

Сбор и экспорт данных из Unreal Engine

В исходном проекте сцена в *Unreal Engine* анализируется с использованием встроенного Python API. Программа, написанная на Python, извлекает ключевую информацию об объектах уровня: названия, классы, трансформации и пути к сеткам, после чего данные сохраняются в формате JSON (см. листинг 1).

Листинг 1. Пример экспорта данных в JSON

```python
# Пример скрипта на Python для экспорта данных из Unreal Engine в JSON
import unreal
import json

def export_level_data(output_file):
    level_actors = unreal.EditorLevelLibrary.get_all_level_actors()
    scene_data = []
    for actor in level_actors:
        actor_data = {
            "name": actor.get_name(),
            "class": actor.get_class().get_name(),
            "transform": {
                "location": actor.get_actor_location().to_tuple(),
                "rotation": actor.get_actor_rotation().to_tuple(),
                "scale": actor.get_actor_scale3d().to_tuple()
            }
        }
        scene_data.append(actor_data)
    with open(output_file, 'w', encoding='utf-8') as f:
        json.dump(scene_data, f, indent=4)

# Пример вызова функции
export_level_data("C:/Export/scene_data.json")
```

Такой формат позволяет структурировать информацию об иерархии объектов, материалах, коллайдерах и прочих атрибутах, необходимых для последующего восстановления сцены в целевом движке.

Преобразование ассетов

Одной из существенных проблем является несовместимость форматов ассетов: *Unreal Engine* использует формат .uasset, который применим исключительно внутри его среды. Для обеспечения совместимости с *Unigine* необходимо конвертировать эти файлы. В предлагаемом решении реализован метод, который преобразует .uasset в .fbx (см. листинг 2), что позволяет сохранить геометрию и анимационные данные, а также обеспечивает возможность повторного импорта в *Unigine*.

Дополнительно производится пересчет касательного пространства (т. е. параметров нормалей и касательных в соответствии с требованиями целевого движка).

Листинг 2. Пример конвертации ассета

```
# Пример функции для конвертации файла .uasset в .fbx
import unreal

def convert_uasset_to_fbx(uasset_path, output_fbx_path):
    # Предполагается, что в Unreal Engine имеется вызов метода конвертации,
    # доступного через Python API (данный код иллюстративный)
    converter = unreal.AssetToolsHelpers.get_asset_tools()
    success = converter.export_asset(uasset_path, output_fbx_path)
    if success:
        unreal.log("Конвертация завершена: " + output_fbx_path)
    else:
        unreal.log_error("Ошибка конвертации: " + uasset_path)

# Пример вызова функции
convert_uasset_to_fbx("C:/Project/Content/MyAsset.uasset", "C:/Export/MyAsset.fbx")
```

Импорт данных в Unigine

После экспорта JSON-файла и конвертации ассетов следующим шагом является импорт полученных данных в *Unigine*. Для этого разработан графический интерфейс, позволяющий указать пути к проектам *Unreal Engine* и *Unigine*, а также автоматизировать перенос ассетов. Программа анализирует JSON-файл, извлекает информацию об объектах, а затем создаёт соответствующие узлы в файле сцены (.world) *Unigine*, корректируя трансформации с учётом различий в системах координат между движками. Особое внимание уделяется корректному восстановлению иерархии сцены, а также обеспечению согласованности параметров освещения и материалов.

При переносе возникает весь спектр ранее описанных трудностей:

- *Несовместимость форматов* – необходимость конвертации .uasset в .fbx и последующая адаптация ассетов под систему *Unigine*.

- *Различия в системах координат* – автоматическая корректировка положений и поворотов объектов для соблюдения новых правил преобразования (ср. рисунки 1 и 2).
- *Пересчёт касательного пространства* – для корректного отображения шейдеров и нормалей требуется адаптация касательного пространства.
- *Сохранение сцены* – восстановление полной иерархии и атрибутов объектов из JSON-файла для точного воспроизведения сцены в новом движке.

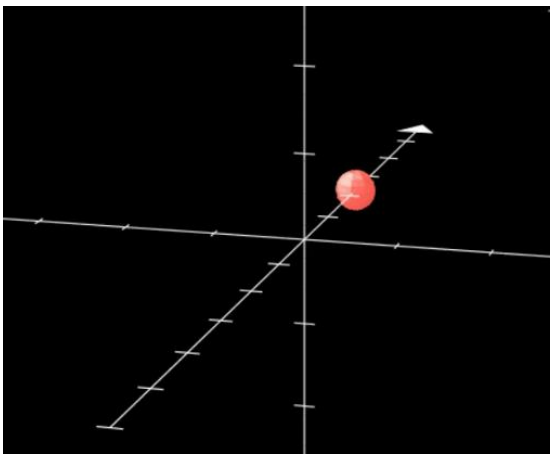

Рис. 1. Система координат в *Unreal Engine*

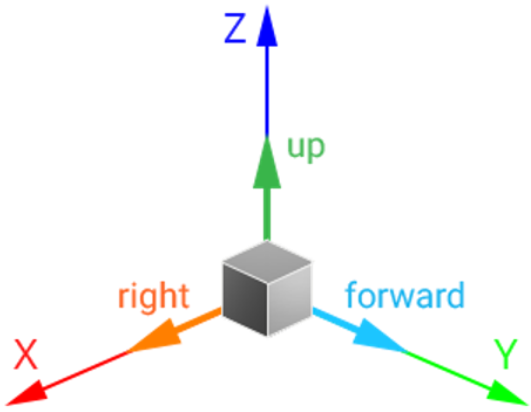

Рис. 2. Система координат в *Unigine*

Таким образом, перенос игровых сцен из *Unreal Engine* в *Unigine* является наглядным примером сложностей миграции контента между игровыми движками (ср. рисунки 3 и 4).

Рис. 3. Тестовая игровая сцена в *Unreal Engine*

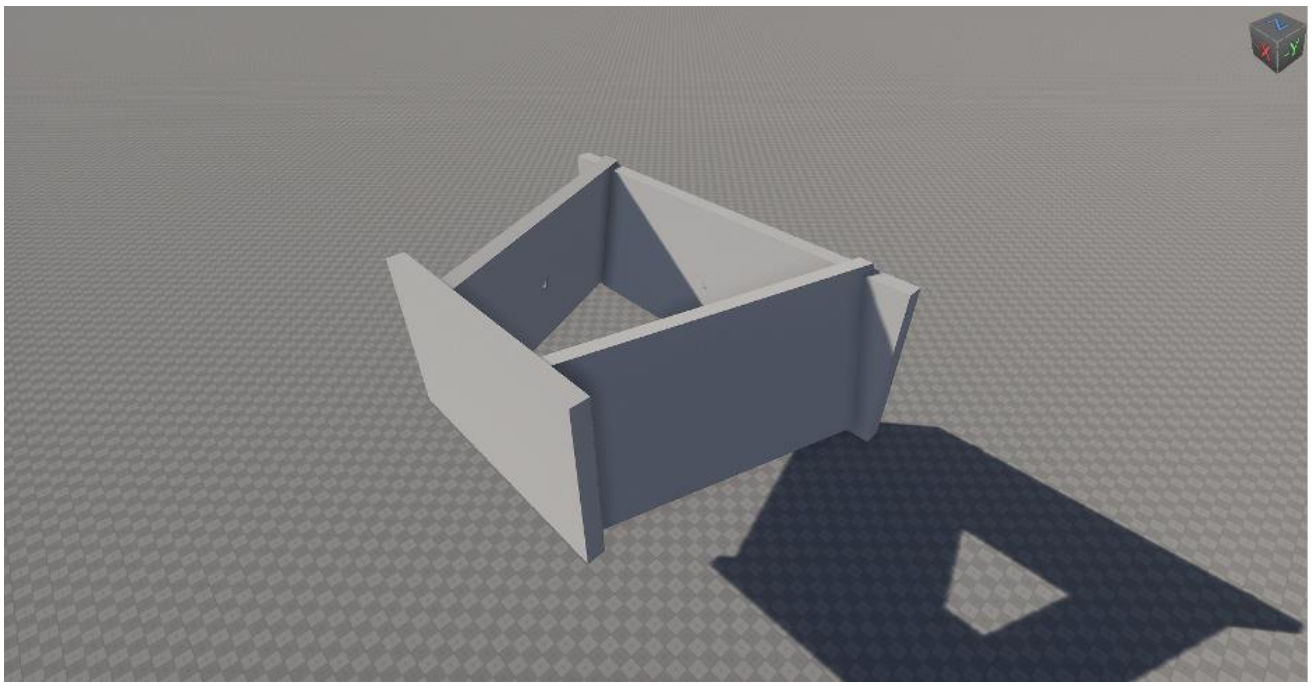

Рис. 4. Перенесенная в *Unigine* тестовая игровая сцена

Реализованные решения демонстрируют, что использование промежуточных форматов (JSON, FBX) и специализированных конвертеров может существенно облегчить задачу, однако окончательная адаптация часто требует значительных усилий и ручной доработки.

Будущие направления автоматизации переноса игровых сцен

Несмотря на значительный прогресс в разработке инструментов для экспорта и импорта игровых сцен, полный перенос контента «под ключ» остаётся масштабной задачей, требующей дальнейших исследований и совершенствования методов.

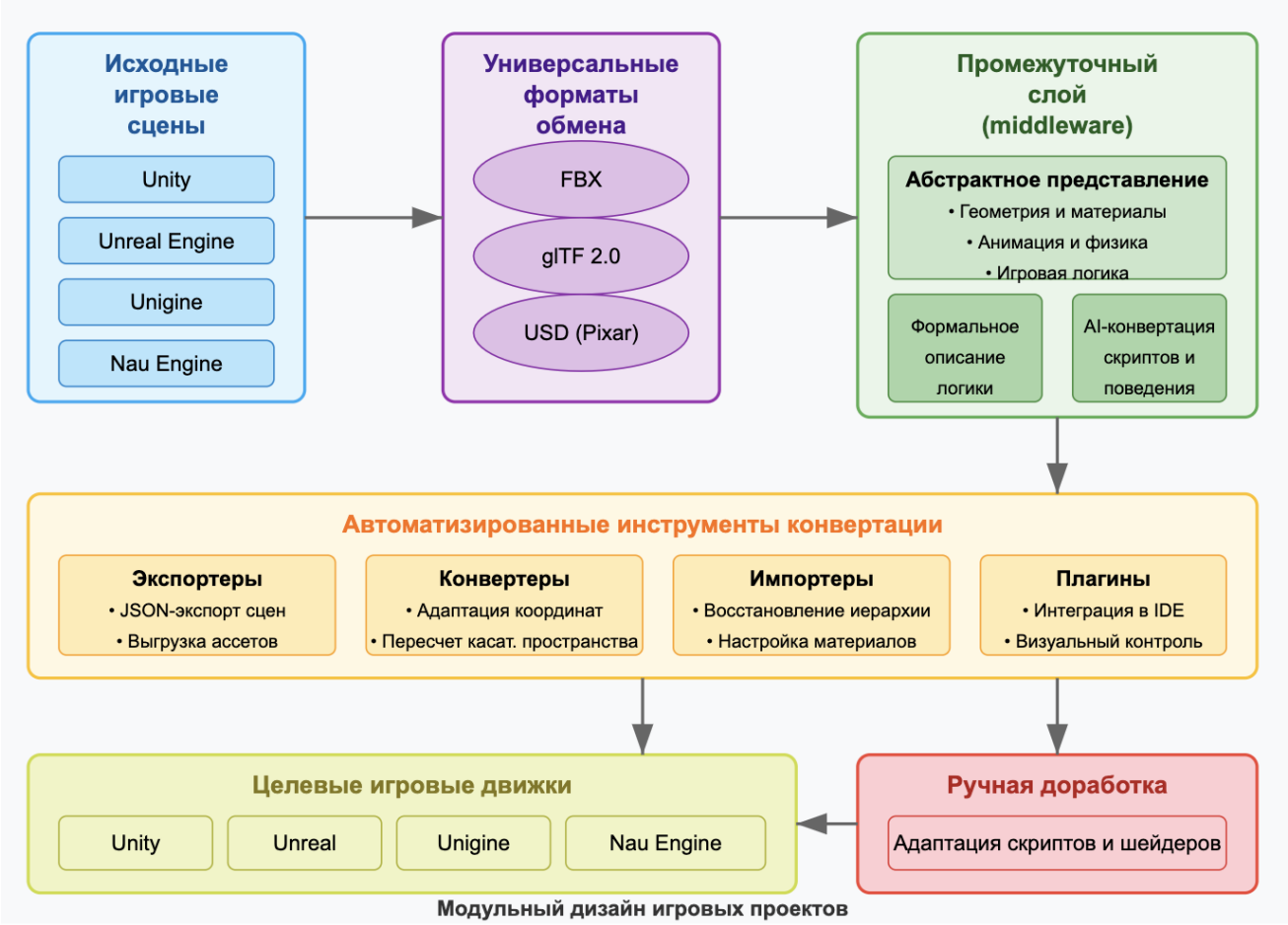

Рис. 5. Ключевые этапы и направления дальнейших исследований

В этом контексте можно выделить следующие перспективные направления (рис. 5).

1. **Унификация форматов обмена данными.**
   Разработка и внедрение универсальных форматов, таких как USD или расширенные версии glTF, способных полноценно описывать геометрию, материалы, анимацию и даже логику игровых сцен, позволяет снизить зависимость от проприетарных решений. Стандартизация обмена данными

между движками является ключевым шагом к автоматизации переноса.

2. **Разработка промежуточного слоя (middleware).**
   Создание абстрактного представления игровых сцен, которое отделяет содержание (ассеты, анимации, логику) от платформенной реализации, может значительно упростить процесс миграции. Такой слой позволит адаптировать одни и те же данные под требования разных движков посредством набора адаптеров.
3. **Автоматизация переноса игровой логики.**
   На данный момент статические данные (геометрия, материалы, анимация) переносятся относительно успешно, тогда как перенос кода и логики требует значительной ручной работы. Перспективным направлением является использование методов искусственного интеллекта для автоматической конвертации скриптов и описания поведения объектов, что позволит автоматизировать и эту часть процесса. Возможно, это необходимо сочетать с промежуточным этапом формального описания игровой логики.
4. **Модульный дизайн игровых проектов.**
   Планирование архитектуры игр с учетом возможности будущей миграции может существенно упростить перенос. Разделение проекта на независимые модули, где логика, визуальные ассеты и физические параметры обрабатываются отдельно, позволит переносить лишь отдельные компоненты, минимизируя усилия при адаптации под новый движок.
5. **Интеграция в рабочие конвейеры разработки.**
   Разработка плагинов и утилит, которые легко интегрируются в существующие системы разработки, такие как *Unity*, *Unreal Engine*, *Unigine* и *Nau Engine*, позволит автоматизировать не только этап переноса, но и его последующую адаптацию что, в свою очередь,позволит разработчикам быстрее переключаться между платформами при изменении требований проекта.

В рамках дальнейших исследований планируется разработать универсальное решение для переноса игровых сцен, которое будет поддерживать миграцию между *Unity*, *Unreal Engine*, *Unigine* и *Nau Engine*. При этом особое внимание будет уделено сохранению заложенной игровой логики, что является особенно важным

для проектов с богатым интерактивным функционалом. Достижение этой цели потребует глубокого анализа специфики каждого движка, создания гибкой архитектуры промежуточного представления данных и разработки адаптивных конвертеров для автоматического преобразования контента.

ЗАКЛЮЧЕНИЕ

В статье дан всесторонний анализ технических аспектов переноса игровых сцен между различными игровыми движками. Рассмотрены основные проблемы, связанные с несовместимостью форматов данных, различиями в API для рендеринга, физики и логики, а также сложностями конвертации материалов, шейдеров и анимационных данных. Изучение существующих инструментов (таких как Unity→Unreal, Unity→Unigine, Unity→Godot) показало, что автоматизация статических аспектов сцены уже достигла заметного прогресса. Однако динамическая логика и интерактивные элементы остаются областью, требующей значительной ручной доработки.

Особое внимание уделено фундаментальным подходам к решению проблемы: использование универсальных форматов обмена (FBX, glTF, USD), разработка промежуточных слоев для абстрагирования данных от специфики конкретного движка и применение модульного дизайна игровых сцен. Эти направления открывают возможности для создания комплексного решения «под ключ», способного обеспечить автоматизированный перенос не только графических, но и логических компонентов игровых проектов.

В качестве примера практической реализации рассмотрен перенос игровых сцен из *Unreal Engine* в *Unigine*, для которого разработан собственный pipeline с использованием Python-скриптов. В дальнейших исследованиях планируется расширение функционала переносчика для поддержки миграции между *Unity*, *Unreal Engine*, *Unigine* и *Nau Engine* с сохранением заложенной игровой логики. Эти усилия позволят существенно снизить трудозатраты при адаптации проектов под новые платформы и обеспечить их высокое качество.

СПИСОК ЛИТЕРАТУРЫ

1. Export From Unity to Unigine in Seconds // Game From Scratch. 2024.

URL: https://gamefromscratch.com/export-from-unity-to-unigine-in-seconds/

2. *Linietsky J.* Why we should all support glTF 2.0 as THE standard asset exchange format for game engines // GodotEngine. 2017.
URL: https://godotengine.org/article/we-should-all-use-gltf-20-export-3d-assets-game-engines/

3. @Coldwalker37. Creating a Unity to Godot converter, and a Unity to Unreal converter // Reddit. GameDev. 2023. URL: https://www.reddit.com/r/gamedev/comments/16j7xc8/creating_a_unity_to_godot_converter_and_a_unity/

4. @NegInfinity. Project Exodus – Unity to unreal scene converter //Unreal Engine. Forum. Plugins. 2019. URL: https://forums.unrealengine.com/t/plugin-project-exodus-unity-to-unreal-scene-converter/125362

5. Quevillon A. Utu Plugin – Unity to Unreal Project Converter. 2023. URL: https://alexquevillon.gumroad.com/l/UtuPlugin

6. Migrating to UNIGINE from Unity: Content Creation // Unigine. URL: https://developer.unigine.com/en/docs/future/migration/from_unity/content

7. *Zylann M.* Unity Engine to Godot Engine exporter // GitHub. 2019. URL: https://github.com/Zylann/unity_to_godot_converter

8. OpenGEX. 2022. URL: https://opengex.org/

9. *Torres J.C.* Star citizen space sim moves to amazon lumberyard game engine // Lash Gear. Gaming. 2016. URL: https://www.slashgear.com/star-citizen-space-sim-moves-to-amazon-lumberyard-game-engine-26468764/

10. Unity to Unreal Engine Game Transfer // Pungle Studio.
URL: https://pinglestudio.com/service/unity-to-unreal-engine-transfer

11. How to transfer your level from Unity3D to Unreal Engine 5 (and vice versa) // Youtube. iBrews. 2023.
URL: https://www.youtube.com/watch?v=H9cN0m-p8zM

12. *Ciekanowska A., Kiszczak-Gliński A., Dziedzic K.* Comparative analysis of Unity and Unreal Engine efficiency in creating virtual exhibitions of 3D scanned models // Journal of Computer Sciences Institute. 2021. Vol. 20. P. 247–253.

13. *Шараева Р.А. и др.* Подходы к проектированию виртуальных тренажёров хирургических операций // Электронные библиотеки. 2022. Т. 25. №. 5. С. 489–532.

14. *Selva P.E.* GameEngine as a SceneIndex plugin // Alliance for OpenUSD. 2024. URL: https://forum.aousd.org/t/gameengine-as-a-sceneindex-plugin/1238

15. *Shikin B.* Migrating from Unity to Other Game Engines // AppLovin. 2023. URL: https://www.applovin.com/blog/migrating-from-unity-to-other-game-engines/

16. *McConchie J., Ensom T.* Preserving virtual reality artworks: a museum perspective // ACM SIGGRAPH 2019 Talks. 2019. P. 1–2.

17. *Kugurakova V.V.* A formal approach to spatio-temporal modeling of game systems // Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki. 2024. V. 166. № 4. P. 532–554.
https://doi.org/10.26907/2541-7746.2024.4.532-554.

18. *Kugurakova V., Vasilov T., Khafizov M.* Approaches to automating VR applications porting using common techniques // BIO Web of Conferences. 2024. Vol. 84. Art. No. 02016.

19. *Бондарь А.* UE to Unigine Exporter // GitHub. 2025.
URL: https://github.com/Bonndii/UEtoUnigineExporter

PROBLEMS, SOLUTIONS AND PERSPECTIVES OF AUTOMATED TRANSFER OF GAME SCENES BETWEEN GAME ENGINES

A.O. Bondar[1] [0009-0005-5296-9686], V.V. Kugurakova[2] [0000-0002-1552-4910]

[1, 2]*Institute of Information Technologies and Intelligent Systems, Kazan Federal University*

[1]alexey.bondar.2013@mail.ru, [2]vlada.kugurakova@gmail.com

Abstract

This article examines the technical challenges involved in transferring game scenes between various game engines. It analyzes the key issues arising from differences in scene formats, incompatibilities in rendering and physics APIs, as well as problems in converting materials, shaders, and animation data, and discrepancies in coordinate systems. Existing tools and methods, including automated solutions for exporting, converting, and importing data, are presented with a particular focus on migrating content from Unreal Engine to Unigine. Furthermore, the paper discusses fundamental approaches to solving the problem, such as the use of universal exchange formats (FBX, glTF, USD), the development of middleware, and the modular design of game scenes, which pave the way for future automation. The work also highlights our group's research results on the formal description of game logic and approaches to porting VR applications across different libraries. The conclusions provide practical recommendations for developers and outline future research directions in the area of automated content transfer between game engines.

Keywords: *game scene migration, game engine, content migration, Unreal Engine, Unity, Unigine, Nau Engine, Godot, CryEngine, format conversion, data standardization.*

REFERENCES

1. Export from Unity to Unigine in Seconds // Game From Scratch. 2024. URL: https://gamefromscratch.com/export-from-unity-to-unigine-in-seconds/

2. *Linietsky J.* Why we should all support glTF 2.0 as THE standard asset exchange format for game engines // GodotEngine. 2017. URL: https://godotengine.org/article/we-should-all-use-gltf-20-export-3d-assets-game-engines/

3. @Coldwalker37. Creating a Unity to Godot converter, and a Unity to Unreal converter // Reddit. GameDev. 2023.
URL: https://www.reddit.com/r/gamedev/comments/16j7xc8/creating_a_unity_to_godot_converter_and_a_unity/

4. @NegInfinity. Project Exodus – Unity to unreal scene converter // Unreal Engine. Forum. Plugins. 2019. URL: https://forums.unrealengine.com/t/plugin-project-exodus-unity-to-unreal-scene-converter/125362

5. *Quevillon A.* Utu Plugin – Unity to Unreal Project Converter. 2023.
URL: https://alexquevillon.gumroad.com/l/UtuPlugin

6. Migrating to UNIGINE from Unity: Content Creation // Unigine.
URL: https://developer.unigine.com/en/docs/future/migration/from_unity/content

7. *Zylann M.* Unity Engine to Godot Engine exporter // GitHub. 2019.
URL: https://github.com/Zylann/unity_to_godot_converter

8. OpenGEX. 2022. URL: https://opengex.org/

9. *Torres J.C.* Star citizen space sim moves to amazon lumberyard game engine // Lash Gear. Gaming. 2016. URL: https://www.slashgear.com/star-citizen-space-sim-moves-to-amazon-lumberyard-game-engine-26468764/

10. Unity to Unreal Engine Game Transfer // Pungle Studio.
URL: https://pinglestudio.com/service/unity-to-unreal-engine-transfer

11. How to transfer your level from Unity3D to Unreal Engine 5 (and vice versa) // Youtube. iBrews. 2023.
URL: https://www.youtube.com/watch?v=H9cN0m-p8zM

12. *Ciekanowska A., Kiszczak-Gliński A., Dziedzic K.* Comparative analysis of Unity and Unreal Engine efficiency in creating virtual exhibitions of 3D scanned models // Journal of Computer Sciences Institute. 2021. Vol. 20. P. 247–253.

13. *Sharaeva A. et al. Approaches to the development of virtual surgical training* // Russian Digital Libraries. 2022. V. 25. No. 5. P. 489–532.
https://doi.org/10.26907/1562-5419-2022-25-5-489-532 (In Russian).

14. *Selva P.E.* Game Engine as a Scene Index plugin // Alliance for OpenUSD. 2024. URL: https://forum.aousd.org/t/gameengine-as-a-sceneindex-plugin/1238

15. *Shikin B.* Migrating from Unity to Other Game Engines // AppLovin. 2023. URL: https://www.applovin.com/blog/migrating-from-unity-to-other-game-engines/

16. *McConchie J., Ensom T.* Preserving virtual reality artworks: a museum perspective // ACM SIGGRAPH 2019 Talks. 2019. P. 1–2.

17. *Kugurakova V.V.* A formal approach to spatio-temporal modeling of game systems // Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki. 2024. Vol. 166. No. 4. P. 532–554. https://doi.org/10.26907/2541-7746.2024.4.532-554.

18. *Kugurakova V., Vasilov T., Khafizov M.* Approaches to automating VR applications porting using common techniques // BIO Web of Conferences. 2024. Vol. 84. Art. No. 02016.

19. *Bondar A.* UE to Unigine Exporter // GitHub. 2025. URL: https://github.com/Bonndii/UEtoUnigineExporter

СВЕДЕНИЯ ОБ АВТОРАХ

БОНДАРЬ Алексей Олегович – магистрант Институт информационных технологий и интеллектуальных систем Казанского федерального университета. Направление исследований: миграция игровых сцен между платформами и игровыми движками.

Alexey Olegovich BONDAR – Master's student at the Institute of Information Technologies and Intelligent Systems of Kazan Federal University. Research area: migration of game scenes between platforms and game engines.

email: alexey.bondar.2013@mail.ru

ORCID: 0009-0005-5296-9686

КУГУРАКОВА Влада Владимировна – доцент, кандидат технических наук, зав. кафедрой индустрии разработки видеоигр Института ИТИС КФУ. Зона научных интересов: формальная верификация видеоигр, иммерсивность виртуальных миров.

Vlada Vladimirovna KUGURAKOVA – Ph. D. of Engineering Sciences, Head of the Video Game Development Industry Department of ITIS KFU.

email: vlada.kugurakova@gmail.com

ORCID: 0000-0002-1552-4910

Материал поступил в редакцию 30 января 2025 года