Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт математики и компьютерных наук
Кафедра алгебры и дискретной математики

3D-визуализация в системе проведения виртуальных соревнований по робототехнике CVARC

Допущен к защите

_____

"__" _____ 2013

Квалификационная работа на степень бакалавра наук по направлению «Математика. Компьютерные науки»
студента гр. КН – 402
Кропотова Максима Сергеевича

_____

Научный руководитель
Окуловский Юрий Сергеевич
заведующий лабораторией РВИИМАП, к.ф.-м.н.

_____

Екатеринбург
2013

РЕФЕРАТ

Кропотов М.С. 3D-ВИЗУАЛИЗАЦИЯ В СИСТЕМЕ ПРОВЕДЕНИЯ ВИРТУАЛЬНЫХ СОРЕВНОВАНИЙ ПО РОБОТОТЕХНИКЕ CVARC, квалификационная работа на степень бакалавра наук: стр. 34, библ. 16 назв.

Ключевые слова: Робототехника, автономные мобильные роботы, трехмерная графика, рендеринг в реальном времени, Microsoft .NET, C#.

В работе представлена высокоуровневая библиотека трехмерной графики реального времени. Требования были сформулированы в ходе разработки системы проведения виртуальных соревнований по программированию роботов CVARC; после анализа существующих решений было принято решение реализовать собственную библиотеку.

Разработан фреймворк, который позволяет по декларативному описанию трехмерной сцены легко получать качественное трехмерное изображение в реальном времени. Фреймворк был успешно применен в CVARC и при условии доработки может применяться в качестве библиотеки 3D-графики общего назначения в различных других проектах.

Содержание

Перечень принятых в работе сокращений 3

Введение 4

Глава 1. Обзор существующих решений 7

Глава 2. CVARC.Graphics: Реализация 11

2.1. Реализм и эффекты 12

2.2. Определение точки обзора: IPointOfView 13

2.3. Сохранение изображений: OffscreenView 16

2.4. Работа с несколькими представлениями 18

Заключение 22

Приложение 1. CVARC: Описание системы 23

Приложение 2. Система тел: простое описание моделируемого мира 29

Литература 33

Перечень принятых в работе сокращений

API — Application Programming Interface (Интерфейс прикладного программирования)

CLR — Common Language Runtime (Общеязыковая исполняющая среда)

CVARC — Competitions in Virtual Autonomous Robots (Соревнования по программированию виртуальных автономных роботов)

IDE — Integrated Development Environment (Интегрированная среда разработки)

OS — Operating System (Операционная система)

Введение

Трехмерная графика является динамично развивающейся отраслью в сфере прикладной разработки программного обеспечения. В течение последних лет качество изображения значительно возросло и стало приближаться к фотореалистичному даже при использовании рендеринга в реальном времени. В большой степени такое развитие связано с индустрией компьютерных игр. Все существующие библиотеки трехмерной графики реального времени основаны на одном из двух API: OpenGL и DirectX. Это устоявшиеся платформы, они поддерживаются всеми современными графическими процессорами и эффективно реализуют всю необходимую для работы с графикой функциональность.

В данной работе представлен фреймворк CVARC.Graphics — высокоуровневая библиотека 3D-графики реального времени, которая позволяет разрабатывать приложения, использующие графику, обладая при этом минимумом специфических знаний. По удобству использования наша библиотека сравнима с ближайшими аналогами и превосходит некоторые из них по производительности.

Необходимость разработки собственной библиотеки возникла в ходе работы над проектом CVARC (системой проведения виртуальных соревнований по программированию автономных роботов), который разрабатывается исследовательским коллективом в Лаборатории Интеллектуальных систем и робототехники ИМКН УрФУ. Цель проекта – популяризация робототехники среди студентов и предоставление им возможности впервые попробовать себя в разработке программного обеспечения для роботов. (Подробнее цели и основные особенности проекта описаны в [1] и приложении 1.) Нами было рассмотрено, как применяется компьютерная графика в различных соревнованиях по программированию и были изучены несколько существующих высокоуровневых

решений для реализации трехмерной графики (более подробное описание в 1). При этом было обнаружено, что специфика CVARC требует от используемой реализации графики возможностей, которые не представлены все вместе в одной библиотеке. В результате было принято решение разработать собственный фреймворк, который получил название CVARC.Graphics.

При этом были выделены основные свойства, которыми должна обладать разрабатываемая библиотека:

- Простота. Использование полученного фреймворка должно требовать от разработчика минимума знаний в области компьютерной графики. Было бы предпочтительно, если бы управлять отрисовкой изображения можно было, используя интуитивно понятные сущности.

- Совместимость с моделью мира, используемой в CVARC. В CVARC используется архитектура MVC [2]. Моделью является виртуальный мир, который задается при помощи дерева тел (подробнее см. в приложении 2) — простых логических сущностей, реализуемых классом Body и его наследниками. Необходимо было сделать, чтобы CVARC.Graphics был трехмерным представлением модели, при этом необходимо было оставить модель независимой от внешних библиотек.

- Возможность сохранения полученных изображений в память. Это требование было вызвано необходимостью эмуляции в CVARC видеокамеры, установленной на робота, которая необходима для того, чтобы участники соревнований могли практиковаться в работе с алгоритмами технического зрения. Для того, чтобы задача работы с изображением была «интересной» (то есть изображение было реалистичным), необходимо было реализовать возможность добавления в него артефактов: бликов, теней, шума, размытия при движении, и т.д.

- Поддержка одновременно нескольких изображений одной и той же трехмерной сцены. При этом должна быть возможность показывать одну и ту же сцену с разных ракурсов, в разных окнах или в частях одного окна.
- Менее приоритетная цель: кроссплатформенность.

В качестве цели работы было принято написание библиотеки, наиболее полно соответствующей указанным требованиям. Нужно отметить, что хотя CVARC.Graphics разрабатывалась для одного вполне определенного применения и не является ни библиотекой 3D-графики общего назначения, ни игровым графическим движком, мы считаем, что она в перспективе может использоваться для различных других проектов. CVARC.Graphics в сочетании с системой тел (классами Body) представляется нам одним из наиболее простых возможных способов определения трехмерных миров.

Глава 1

Обзор существующих решений

Используемые для реализации трехмерной графики технологии значительно зависят от предметной области, в которой они применяются. Так, например, в компьютерных играх и в сфере Big Data Visualization (визуализации больших наборов данных) будут применяться совершенно разные решения. Поэтому для выбора оптимальной технологии и общего подхода к реализации 3D-графики в CVARC, рассмотрим, прежде всего, как графика используется в различных соревнованиях по программированию. Ближайшими аналогами CVARC нам представляются: соревнования по программированию Google AI Challenge, эмулятор роботов USARSim, который используется в качестве средства тестирования в соревнованиях по разработке ПО для автономных роботов Robocup Rescue и Gazebo — популярный эмулятор роботов общего назначения. Рассмотрим подробнее некоторые особенности реализации графики в данных проектах:

- Gazebo для отображения моделируемого мира использует самостоятельно разработанный модуль 3D-графики, использующий OpenGL [3]. Реализован удобный механизм для описания моделируемого мира: объекты простой геометрической формы (шары, параллелепипеды, и т.д.) можно задавать при помощи готовых примитивов, более сложные можно импортировать в виде готовых 3D-моделей в форматах STL или Collada. Реализована возможность получения изображения с камеры, установленной на робота, однако в ней отсутствуют реалистичные искажения, такие, как размытие при движении (Motion Blur).[4]

- В Google AI Challenge используется схематичная двухмерная графика в

браузере, реализованая на «чистом» (т.е. без использования сторонних библиотек) JavaScript[5]. Благодаря этому один и тот же код визуализации может использоваться и на компьютере участника, и на веб-странице соревнований. При этом сохраняется совместимость с любой операционной системой и большинством современных браузеров.

- USARSim предназначен для моделирования сцен с большим количеством сложных объектов с высокой точностью. Поэтому для 3D-графики и моделирования физических взаимодействий в нем используется игровой трехмерный движок Unreal Engine [6]. Такой подход позволяет свести к минимуму временные затраты на разработку, используя при этом все новейшие достижения в области 3D-графики (т.к. большинство из них активно внедряются прежде всего в компьютерных играх). Основная проблема — Unreal Engine является проприетарным продуктом с закрытым исходным кодом.

В рассмотренных примерах выделяются два подхода:

- самостоятельная разработка библиотеки для 3D-графики на основе одного из распространенных API (DirectX или OpenGL);
- использование готового игрового движка.

Первый подход требует значительных временных затрат, но позволяет получить средство, в точности соответствующее целям конкретного проекта. Второй способ позволяет достичь более высокой степени реализма в изображении при сокращении затрат, но связан с определенным риском. Не понятно, что делать в случае, если в процессе работы будет обнаружен недостающий функционал: дорабатывать чужой низкоуровневый код зачастую сложно; ожидание новой версии движка от его разработчиков может занять неопределенный срок.

Для полноты картины необходимо также упомянуть некоторые библиотеки, возможность применения которых мы рассматривали:

- WPF — интегрированное решение для разработки интерфейсов, входяще в состав Microsoft .NET. Одним из модулей является библиотека 3D-графики, которая поддерживает возможность простого создания 3D-объектов. Моделируемую сцену можно определять при помощи высокоуровневых сущностей — моделей, не заботясь при этом о рендеринге отдельных кадров (что необходимо делать в DirectX или OpenGL). WPF использовалась нами в проекте Eurosim [7] (прототипе CVARC), впоследствии было принято решение от ее применения отказаться. Основными проблемами, которые были обнаружены нами в ходе работы и сделали дальнейшее использование данной технологии нецелесообразным, были: отсутствие поддержки аппаратного ускорения, как следствие, очень низкая производительность; невозможность рендеринга изображения в файл. При этом нужно отметить, что декларативный способ задания трехмерной сцены в WPF очень удобен и послужил одним из прототипов для системы тел (см. описание в приложении 2) в CVARC.

- WebGL — технология для отображения 3D-графики в веб-страницах с использованием JavaScript и поддержкой аппаратного ускорения. К сожалению, WebGL не была нам известна на момент начала работы над CVARC и поэтому не рассматривалась. Впоследствии с использованием данной технологии была реализована экспериментальная версия просмотрщика записей матчей. Технология кажется нам перспективной в силу того, что не требует от пользователей установки какого-либо ПО; мы предполагаем, что в будущем при помощи нее можно будет полностью заменить отдельное приложение для проигрывания реплеев. Полностью заменить существующую реализацию 3D-графики не получится

из-за необходимости генерировать изображения с виртуальной камеры на сервере — для них должно использоваться решение на основе DirectX или OpenGL. Также в настоящее время WebGL не поддерживается ни одной из версий браузера Internet Explorer [8], что является значительным ограничением.

Рассмотрев существующие аналоги, мы приняли решение разработать для использования в CVARC собственную высокоуровневую библиотеку 3D-графики. В качестве основы был выбран SlimDX [9] — «обертка» над API DirectX, позволяющая использовать всю его функциональность из среды Microsoft .NET. Реализованная в результате CVARC.Graphics в некотором смысле является аналогом 3D-модуля в WPF. Хотя CVARC.Graphics и WPF разрабатывались исходя из совершенно разных соображений, у них есть общий принцип: они обе предназначены для максимально простого описания трехмерных сцен, при этом обе предоставляют доступ к некоторому подмножеству DirectX.

Глава 2

CVARC.Graphics: Реализация

Основная особенность библиотеки CVARC.Graphics — прозрачность для разработчика, который использует ее для отображения трехмерной графики в своем приложении. Разработчик определяет модель — это декларативное описание составляющих сцену объектов — при помощи системы тел (см. приложение 2) и один раз при инициализации передает ее библиотеке графики. Для перемещения объектов, или изменения их свойств, таких, как цвет или положение в пространстве, требуется только изменять соответствующие свойства модели. При таком подходе от разработчика не требуется знания каких-либо понятий из области графики, вместо этого он может оперировать простыми логическими сущностями.

Основной объект CVARC.Graphics — представление (View). Реализованы два вида представлений: FormView отображает изображение на экран, а OffscreenView (см. 2.3) — сохраняет в память. Для удобства добавления представлений внутрь существующего интерфейса реализованы классы-адаптеры [10] FormWrapper (наследник System.Windows.Forms.Form) и ControlWrapper (наследник System.Windows.Forms.Control).

Пример: предположим, что разработчик приложения определил модель некоторой 3D-сцены в виде дерева тел и сохранил ее корень в переменную model. Тогда для того, чтобы создать окно с изображением сцены ему достаточно написать следующий код:

```
    var scene = new Scene(model); //Создается сцена.
    var view = new FormView(scene); //Создается представление.
    var form = new FormWrapper(view); //Создается форма –адаптер для представления.
    Application.Run(form);
    //Запускается приложение, у которого form является основным окном.
}
```

Если изображение требуется добавить в интерфейс существующего приложения, то последние две строчки можно заменить на parentControl.Add(new ControlWrapper(view)), где parentControl — элемент интерфейса, в котором следует показывать изображение. ControlWrapper — это класс-адаптер для представления, представляющий собой элемент интерфейса Windows Forms. Разумеется, здесь приведен самый простой пример: такой код инициализирует представление с настройками по умолчанию. У представления есть различные дополнительные свойства, из них два наиболее важных: Effect — позволяет определить эффект, способ отрисовки (см. 2.1) и PointOfView — задает используемую точку обзора (см. 2.2). Также свойства представления позволяют управлять размерами изображения на экране, конфигурацией источников света, и т.д.

2.1. Реализм и эффекты

Одно из основных требований, предъявляемых CVARC к используемой реализации 3D-графики — высокое качество изображения. В частности — необходимо было, чтобы в представлении, генерирующем изображения для эмуляции камеры на роботе присутствовали артефакты, затрудняющие его анализ с помощью алгоритмов компьютерного зрения: шум, размытие при движении. При этом в представлении, используемом для отображения на экран вида на игровое поле, очевидно, подобные искажения неуместны; вместо этого необходимо создать иллюзию «объемного» вида сверху, что дости-

гается при помощи использования теней. Так возникает необходимость выделения способа отрисовки в отдельную сущность, отдельную от сцены и представлений (т.к. одна сцена может отображаться в разных представлениях с разными эффектами). Эту задачу решает абстрактный класс Effect. Эффект — это определенный способ отрисовки сцены. Оба вида реализованных представлений позволяют устанавливать используемый Effect при помощи одноименного свойства. На данный момент реализован единственный эффект — DefaultEffect, который рисует сцену с простейшими планарными тенями. Примеры возможных эффектов, которые планируется реализовать в будущем: эффект, добавляющий в изображение размытие при движении; эффект, синтезирующий на основе сцены карту глубин.

2.2. Определение точки обзора: IPointOfView

«Пирамидой видимости» (view frustum) в 3D-графике называется область трехмерной сцены, отображаемая на плоском экране; для ее задания, как правило, используются три вектора: CameraPosition, CameraTarget, CameraUpVector [11]. При этом на практике при использовании API DirectX для установки области видимости необходимо передать определенным образом посчитанную матрицу. Хотя операция генерации матрицы по трем векторам в DirectX реализована, и один раз посчитать необходимую матрицу не представляет труда, такой подход делает любое перемещение области видимости трудоемкой задачей.

Напомним, что наша цель — разработка удобной высокоуровневой библиотеки, не требующей от пользователя специфических знаний в области компьютерной графики. Поэтому более естественным кажется управление обзором при помощи сущности «камера» (или «точка обзора»). Удобно было бы предоставлять пользователю несколько го-

товых «камер»: «вид сверху», «вид от первого лица», «вид от третьего лица» с возможностью вращать камеру вокруг заданной точки.

```
public interface IPointOfView
{
  public Matrix ViewTransform { get; }
}
```

Для этого был выделен интерфейс IPointOfView, созданы несколько его реализаций — типичных «точек обзора» и (для удобства) класс-обертка SwitchablePov, который при вызове метода SwitchMode(PovMode mode) переключает вид между ними. Рассмотрим, как при помощи вычисления матрицы вида устанавливается точка обзора в каждой из существующих реализаций интерфейса:

- TopPov — вид сверху. В данном случае камера, находящаяся в точке (0, 0, z) и смотрящяя на плоскость OXY, задается очевидным образом: CameraPosition=(0,0,z), CameraTarget=(0,0,0), CameraUpVector=(0,0,1). Матрица вида тогда вычисляется стандартным методом Matrix.LookAtLH(CameraPosition, CameraTarget, CameraUpVector).

- FirstPersonPov —вид от первого лица, привязанный к заданному телу. Пусть defaultViewMatrix — матрица вида, соответствующая камере, находящейся в начале координат и смотрящей параллельно оси OX в ее положительном направлении. Тогда если $\overrightarrow{Coords}$ (вектор из 6 координат: X,Y,Z,Yaw,Pitch,Roll) — абсолютные координаты тела, на которое установлена камера, а $\overrightarrow{Offset}$ — смещение камеры относительно тела, то матрица вида для точки обзора от первого лица вычисляется следующим образом:

$$ViewMatrix = MovementMatrix(\overrightarrow{Coords}) \times (MovementMatrix(\overrightarrow{Offset}) \times defaultviewMatrix)$$

- FirstPersonPov — реализует камеру, вращающуюся вокруг начала координат. Управление такой камерой осуществляется при помощи естественного «вращения» объектов мышкой при нажатой левой кнопке. Для этого в классе FirstPersonPov используются обработчики событий OnMouseDown, OnMouseUp и OnMouseMove Поскольку при вращении камеры расстояние до начала координат не меняется, также реализована возможность приближения и удаления от него при помощи колесика мышки, для этого используется событие OnMouseWheelZoom.

  Рассмотрим подробнее, как на основе перемещения мышки вычисляется необходимая матрица вида. Пусть начальное положение камеры — (x_0, y_0, z_0); тогда радиус сферы, по которой будет двигаться камера: $r = \sqrt{x_0^2 + y_0^2 + z_0^2}$. В момент времени i мышка переместилась в экранных координатах на (x_{screen_i}, y_{screen_i}, $mouseWheel_i$); ScreenWidth и ScreenHeight — экранные размеры всего изображения. Сначала смасштабируем каждую из координат к промежутку [0;1], учитывая при этом, что экранная ось Y направлена вниз:

$$x_{screen} = \frac{x_{screen}}{ScreenWidth} - 1$$

$$y_{screen} = 1 - \frac{y_{screen}}{ScreenHeight}$$

  Изменение экранной координаты X (движение мышки по горизонтали) соответствует вращению камеры вокруг оси OZ. Экранная координата

Y определяет движение в вертикальной плоскости. Поэтому новые координаты x и z:

$$z_i = y_{screen} \times (z_{i-1} + 1)$$

$$x_i = \sqrt{r - z_i^2}$$

Если исходный масштаб изображения равен 1, то масштаб в момент времени i равен:

$$Scale_i = \prod_{j=1}^{i} e^{\frac{\Delta_j}{ScreenHeight}}$$

Для учета масштаба разделим на него координаты x_i, y_i, z_i. Посчитаем матрицу вида с новыми координатами при помощи стандартного метода и домножим её слева на матрицу, соответствующую вращению вокруг OZ на x_{screen} (это вычисляется при помощи стандартного метода Matrix.RotateZ(double angle)). Так по перемещению мышки мы смогли посчитать матрицу вида, соответствующую вращению камеры вокруг начала координат.

Рассмотренная реализация позволила сделать управление ракурсом обзора в CVARC.Graphics тривиальной задачей: для этого всего лишь необходимо в используемом представлении установить свойство PointOfView, точно так же легко настроить чтобы разные представления изображали одну и ту же сцену с разных сторон.

2.3. Сохранение изображений: OffscreenView

Для сохранения изображений реализован отдельный вид представления — OffscreenView. Конструктор представления, как и в случае с обычным представлением, принимает сцену, которая содержит модели объектов. Единственный метод GetImage возвращает изображение; также есть возможность

определять точку обзора (см. 2.2). Предполагалось, что при использовании данного представления в CVARC будут реализованы различные искажения в изображении, например, размытие при движении, шумы, и т.д. Такие искажения не были реализованы, но их использование предусмотрено — для этого достаточно установить поле Effect (см. 2.1).

```
public class OffscreenView
{
  public OffscreenView(Scene scene, int width, int height)
  {
  /*...*/
  }

  public byte[] GetImage()
  {
  /*...*/
  }

  public ImageFileFormat imageFileFormat { get; set; }
  public PointOfView PointOfView {get; set; }
  public Effect Effect { get; set; }
}
```

Реализация такого представления несколько отличается от случая, когда изображение показывается на экране: вместо SwapChain здесь создается RenderTarget, не привязанный к Window Handle. После рендеринга сцены содержимое Render Target полностью копируется в Offscreen Surface, затем сохраняется в массив байт в заданном формате. Поддерживаются все основные форматы изображений (Jpeg, Bmp, Tiff, и т.д.)

Описанный алгоритм при размере 800 × 600 пикселей и формате jpeg позволяет получать изображение с частотой около 15 кадров в секунду. (Для сравнения: при тестировании на том же аппаратном обеспечении при рендеринге на экран можно получить частоты от 50 до 100 кадров в секунду). При увеличении размера изображения максимально возможная частота уменьшается. Мы предполагаем, что такая низкая производительность связана с необ-

ходимостью копирования данных об изображении из видеопамяти в оперативную память. Существуют альтернативные способы решения данной задачи, например, техника Render-to-Texture [11], но они показывают еще худшую производительность. Существующая производительность удовлетворяет требованиям CVARC, где OffscreenView используется для эмуляции видеокамеры, установленной на робота, а частота получения данных с сенсоров ограничена 10.

2.4. Работа с несколькими представлениями

В CVARC.Graphics реализована возможность использования нескольких представлений одновременно. Классы-адаптеры FormWrapper и ControlWrapper позволяют использовать представление для отображения на экран как в отдельном окне, так и внутри элементов интерфейса. Так можно создать несколько представлений, а передав им разные PointOfView или Effect — получить возможность визуализировать одну и ту же сцену с разных ракурсов или с разными эффектами. Также реализовано представление, сохраняющее изображение в файл.

Для изображения некоторой 3D сцены в окне или его части в DirectX используется объект SwapChain, содержащий Front Buffer (первичный буфер) и Back Buffer (вторичный буфер) — области памяти, в которых вычисляется изображение соответствующее именно этой области экрана. Каждое представление, отрисовывающее сцену на экран, создает собственный SwapChain и работает с ним, обновляя изображение по таймеру в соответствии с частотой смены кадров.

Наибольшие трудности при реализации работы с несколькими представлениями были вызваны ограничениями, которые накладывает DirectX:

1. DirectX реализован при помощи неуправляемого кода (Unmanaged Code),

поэтому при работе с его объектами из среды CLR необходимо вручную освобождать ресурсы при помощи метода Dispose.

2. Большинство методов объектов DirectX непотокобезопасны и требуют использования механизмов блокировки.

3. Device может быть создан только в потоке, обладающем свойством Single-Threaded Apartment State.

4. Методы Device.Reset()(вызывается при изменении размеров) и Device.Dispose() могут быть вызваны только из потока, в котором Device был создан.

5. В конструктор Device должна быть передана форма (объектSystem.Windows.Form у которой должен быть уже инициализирован Window Handle (который создается ОС через некоторое время после вызова конструктора Form).

Получается, что Device нельзя инициализировать ни в одном из потоков:

- В потоке, создающем представления — нельзя, т.к в этот момент еще не созданы используемые впоследствии объекты Form.
- В потоке, обрабатывающем события любого окон — нельзя, т.к. этот поток завершается при закрытии окна.
- В потоках тред-пула — нельзя, т.к. они не являются STAThread.

Проблема была решена введением особого класса DeviceWorker, который отвечает за выполнение «служебных» операций. Для этого DeviceWorker создает выделенный тред и передает ему на исполнение делегат тогда, когда для данного метода есть ограничение по исполняющему его потоку (т.е. необходимо выполнить Create, Reset, Dispose). Инициализация Device происходит «лениво» — когда к нему первый раз обратится какое-либо представление.

```
internal class DeviceWorker
{
  public event Action BeforeReset;
  public event Action AfterReset;
  public event Action Disposing;

  public bool ResetIfRequired(IView view, Size updatingSize) {/* ...*/ }

  public SwapChain GetSwapChain(IView view, Control control) {/* ...*/ }
  public Surface GetRenderSurface(Iview view, Size size) {/* ...*/}

  public void TryDispose(IView view) {/* ...*/ }
}
```

Пример: пусть некоторое приложение использует три представления, два, отображающие некоторую 3D-сцену в окна и третье, сохраняющее изображение в память. Рассмотрим последовательности действий, которые будут выполняться в разных ситуациях.

Изменение размеров:

1: Пользователь уменьшает размер окна 1, при этом срабатывает событие Resize.

2: Представление 1 вызывает метод deviceWorker.ResetIfRequired(/*...*/) передав требуемые размеры. DeviceWorker находит необходимые размеры для Device (минимум из размеров всех представлений). Если эти размеры совпадают со старыми, то меняются размеры только представления 1 и связанной с ним SwapChain, больше никаких операций не выполняется.

3: DeviceWorker поднимает событие BeforeReset. Обработчики события в представлениях 1, 2 и 3 освобождают используемые ресурсы.

4: DeviceWorker в выделенном треде изменяет размеры Device.

5: DeviceWorker поднимает событие AfterReset. Обработчики события в представлениях 1, 2 и 3 создают новые SwapChain с нужными размерами.

Завершение работы:

1: Пользователь закрывает окно 1. Представление 1 освобождает используемые ресурсы и вызывает метод deviceWorker.TryDispose(/∗...∗/)
2: DeviceWorker удаляет представление 1 из списка работающих представлений. При необходимости меняет размер Device так, как описано выше.
3: Пользователь закрывает окно 2. Представление 2 вызывает метод deviceWorker.TryDispose(/∗...∗/).
4: DeviceWorker удаляет представление 2 из списка. Поскольку больше представлений, связанных с окнами нет, то DeviceWorker освобождает используемые ресурсы и вызывает событие Disposing.
5: Оставшееся представление 3 (оно не связано с окном, так как сохраняет изображение в память) в обработчике события Disposing освобождает использовавшиеся им ресурсы.

В результате вынесения всей логики работы с Device в такой класс была значительна упрощена остальная часть нашей библиотеки. Тот факт, что все методы DeviceWorker потокобезопасны позволил избежать использования механизмов синхронизации в представлениях, таким образом упростив код. Также нет никаких ограничений на количество или типы одновременно используемых представлений.

Заключение

В данной работе была представлена реализация библиотеки трехмерной графики CVARC.Graphics. Целью работы было написание средства для трехмерного отображения происходящего в системе проведения виртуальных соревнований по программированию роботов CVARC. Библиотека была успешно интегрирована с остальными компонентами системы и использовалась при проведении первых соревнований (см. приложение 1, [1]). Другие разработчики проекта положительно отзывались о библиотеке графики, отметив, прежде всего, удобство использования.

Хотя поставленная нами цель — разработать решение, совместимое с одним конкретным проектом — была довольно специфичной, мы считаем, что в ходе работы были получены важные результаты, а фреймворк CVARC.Graphics может при условии некоторой доработки использоваться и в других проектах. Ближайший аналог нашей библиотеки — трехмерный модуль из Windows Presentational Foundation (WPF). У обеих библиотек простота использования ставится впереди качества изображения, у каждой есть функциональность, отсутствующая в другой. При этом CVARC.Graphics имеет более высокую производительность: поддерживает большее количество находящихся в одной сцене объектов, чем WPF, а при визуализации одинаковых тестовых сцен показывает значительно меньшую загрузку процессора.

В будущем работу над проектом планируется продолжить. Мы считаем, что для проекта CVARC.Graphics был верно выбран основной принцип — простота использования. Реализована расширяемая архитектура, которая позволит вносить инкрементальные улучшения, без значительных изменений существующих компонентов. Поэтому основное направление развития — улучшение реализма изображения и добавление различных эффектов: шумов в изображении, более реалистичных теней, поддержки зеркальных объектов, и т.д.

Приложение 1

CVARC: Описание системы

CVARC (Competions in Virtual Autonomous Robots)[1] — система для проведения виртуальных соревнований по управлению автономными роботами. Соревнования в области робототехники, такие, как например, Eurobot[12] или Robocup, пользуются популярностью и ежегодно привлекают большое количество участников из университетов разных стран. В ходе подготовки к соревнованиям студенты получают возможность решать наукоемкие задачи в области компьютерных наук, такие как: оптимальное планирование движения, распознавание объектов при помощи технического зрения, и т.д. При этом само участие в соревнованиях связано с большими рисками, материальными затратами, необходимостью работы с аппаратным обеспечением. Такие «элементарные» задачи как обеспечение точного перемещения робота по прямой линии, разработка надежной механики на самом деле нетривиальны и требуют большого количества времени, а изготовление манипулятора необходимой формы может вообще оказаться непреодолимой трудностью для студентов-программистов из-за отсутствия оборудования. При этом, например, в Eurobot соревнования проходят раз в год, команды могут готовиться на протяжении всего года, а продолжительность матча равна 90 секундам. Все это связывает участие с принятием огромных рисков, повышает порог вхождения и делает участие привлекательным только для тех, кто готов делать значительные вложения. Параллельно с этим существует общий тренд к «геймификации» образования. Соревнования ACM позволяют научиться сложным алгоритмам, CTF [13] — навыкам в области компьютерной безопасности. Некоторые научные конференции, такие как ImageCLEF [14], также проводятся в форме Challenge. Также существуют виртуальные соревнования в области робототех-

ники, такие, как Robocup Virtual — в них участникам предлагается разрабатывать алгоритмы управления, но не предполагается, что они будут запущены на реальных роботах; все тестирование происходит с использованием эмулятора. В случае с Robocup Virtual это делается в связи с чрезвычайной сложностью окружения, в котором роботам предлагается решать задачи [6].

Мы предполагаем, что виртуальные соревнования по программированию роботов — идеальное средство для популяризации робототехники и привлечения в нее школьников, студентов, начинающих программистов. Для этого нами был разработан проект CVARC; он представляет собой фреймворк для простой организации таких виртуальных соревнований. В настоящий момент проект находится на стадии рабочего прототипа. Первые соревнования («Пещера великого дракона» или «Suuraz Ypud») были проведены в ИМКН УрФУ в апреле 2013, в рамках празднования Дня математика и механика. В соревнованиях участвовало 6 команд, что позволило проверить работоспособность системы. Мы предполагаем, что небольшое количество участников было связано больше с организационными недочетами, чем с техническими ошибками: в частности, небольшим временем между опубликованием правил и самими соревнованиями. Следующие соревнования планируется провести в октябре 2013 по измененной системе: правила будут публиковаться в несколько этапов (каждый следующий сложнее) с несколькими промежуточными турнирами. Такая схема кажется нам более перспективной.

CVARC представляет собой набор библиотек для платформы Microsoft .NET, написанных на языке C#, необходимых для того, чтобы сторонний разработчик мог организовать соревнования: сервер, клиентское приложение для просмотра реплеев (повторов матчей) и Tutorial - тестовое приложение для демонстрации правил. Организаторы, желающие провести соревнования, должны всего лишь определить правила (расположение объектов на игровом поле и правила подсчета очков), после чего запустить сервер и опубликовать кли-

ентские приложения для участников. Логика обмена данными по сети, эмуляция получения данных с сенсоров, запись повторов и т.д. уже реализованы в библиотеках CVARC и не требуют изменения. Таким образом, процесс проведения соревнований делается максимально простым.

Задача участника — написать приложение, которое управляет виртуальным роботом так, чтобы робот набрал максимальное количество очков в соответствии с правилами соревнований. Побеждает тот, чье приложение набирает наибольшее число очков. Программа участника устанавливает соединение с сервером по протоколу TCP, после чего обменивается с ним сообщениями в формате XML. Сервер генерирует пакеты, содержащие данные с сенсоров, установленных на виртуальном роботе (например, с видеокамеры, Kinect, или навигационных устройств). Программа участника должна обработать данные, принять на основе их решение и ответить пакетом, содержащим команду на перемещение или совершение некоторого действия. По завершению отведенной длины матча сервер посылает реплей — запись всего, что происходило на протяжении матча.

Важно отметить, что CVARC не является эмулятором роботов, таким, как например, Gazebo или USARSim. В робототехнике эмуляторы используются для отладки алгоритмов в условиях, максимально приближенных к реальным, для последующего максимально простого их переноса на реального робота [15] [16]. Для этого в них реализуются точные модели физических устройств (сенсоров, манипуляторов) и реальные протоколы взаимодействия с ними (например, ROS). В отличие от них, CVARC нацелен на начинающих разработчиков, а не на профессионалов, и цель проекта – не разработка промышленного ПО, а популяризация робототехники и привлечение в нее людей. Из этого следуют основные особенности:

- Отсутствие точных моделей устройств, вместо них используются упро-

щенные логические сущности. Например, если в реальном роботе для навигации могут использоваться энкодеры, гироскопы, компас — электронные устройства, каждое со своим низкоуровневым протоколом взаимодействия, то в CVARC они заменяются классом Navigator, который возвращает объект с координатами. При этом координаты могут содержать реалистичный шум (при соответствующих настройках сервера) — поэтому участникам все равно нужно решать «интересную» задачу очистки данных от шума, но не нужно заниматься «техническими моментами» — конвертацией численных данных из специфического для данного сенсора протокола.

- Упрощенное моделирование физических взаимодействий. Для моделирования физики используется двухмерный физический движок (Farseer Physics Engine — стороннее решение с открытым исходным кодом), что позволило упростить код проекта и уменьшить вычислительную нагрузку при работе. Двухмерная физика не накладывает существенных ограничений, т.к. в большинстве соревнований (в том числе, например [12]) роботы перемещаются исключительно на площади. При этом полного отказа от реализма нет — присутствует трение, что усложняет управление роботом — участник должен учитывать, что реальное перемещение будет отличаться от заданного; робот может сталкиваться с объектами, находящимися на игровом поле, толкать их.

- Использование XML для обмена данными. XML почти не используется в реальной робототехнике, вместо этого применяются специфические протоколы, например ROS. Их применение продиктовано необходимостью создания единого стандарта для различного стороннего ПО и устройств. В нашем случае такой необходимости нет, поэтому XML был выбран как наиболее простой протокол, доступный в абсолютном большинстве

языков программирования.

Если обобщить сказанное в один принцип: простота использования (как для организаторов соревнований, так и для участников) приоритетнее точного моделирования всех процессов. Основное преимущество такого подхода — предельно низкий порог вхождения для участников, достаточно лишь начальных знаний в области программирования. Клиент-серверная архитектура позволяет избавиться от ограничений на используемые участниками языки программирования, библиотеки, или ОС. В этом отношении CVARC имеет преимущество перед, например, ACM, где решать задачи можно только на некоторых языках.

Архитектура сервера основана на паттерне MVC [2]. Данный паттерн был разработан преимущественно для веб-приложений, но удивительно удачно подошел в качестве основы для нашего приложения. Модель (Model) содержит бизнес-логику, в нашем случае это совокупность объектов, находящихся на игровом поле. Контроллер (Controller) отвечает за взаимодействие приложения с внешним миром, принимает команды и передает их модели. В CVARC в роли контроллера выступает физический движок и модуль взаимодействия по сети — они оба могут вносить изменения в модель. Представление (View) отображает модель пользователю, в нашем случае им является библиотека 3D-визуализации. Представления не зависят от остальной архитектуры и легко заменяемы.

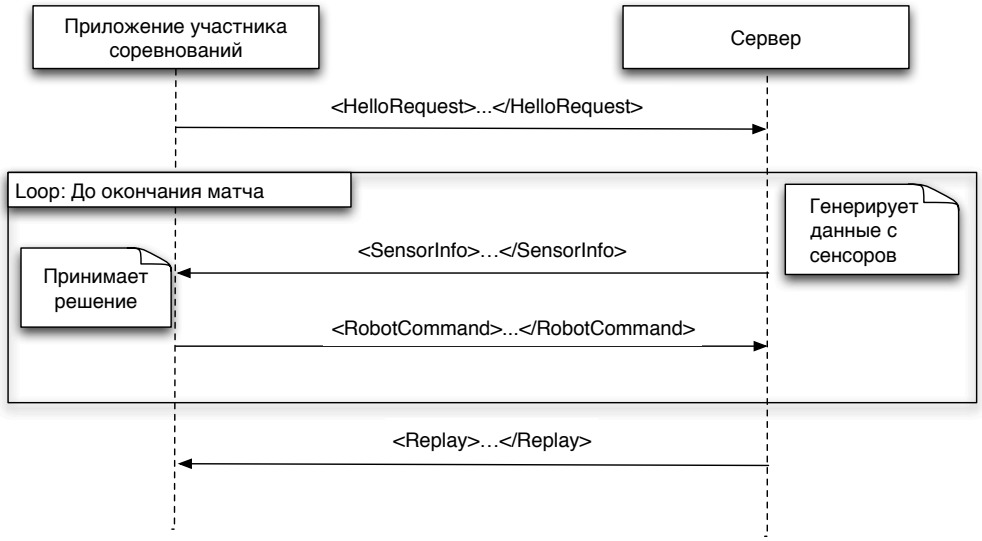

Рис. 1.1. Схема взаимодействия приложения участника и сервера CVARC

Приложение 2

Система тел: простое описание моделируемого мира

Моделью в CVARC является виртуальный мир, в котором происходят соревнования. Мир может содержать:

- роботов;
- неподвижные элементы стола, с которыми роботы могут сталкиваться;
- объекты, которые можно перемещать;
- «нематериальные» декоративные объекты (например, таким образом можно моделировать объекты, находящиеся вокруг игрового поля; они недоступны физически, но видимы и могут влиять на работу технического зрения).

Для описания всего этого многообразия объектов введем понятие тела. Тело — это любая сущность виртуального мира, которая видима или может физически взаимодействовать с другими. Из этого следуют основные свойства, которыми обладают тела: используемые для графического отображения и физические свойства, которые определяют взаимодействия тел. Большинство свойств интуитивно понятны.

```
public class Body{
  /*
  ...
  */
  //Положение в пространстве
  public Frame3D Location { get; set; }
  //Скорость
  public Frame3D Velocity { get; set; }

  //Свойства, используемые для определения физических взаимодействий
  public virtual double Volume { get; }
  public Density Density { get; set; }
  public double FrictionCoefficient { get; set; }
  public bool IsMaterial { get; set; }
  public bool IsStatic { get; set; }

  //Свойства, используемые для определения графического представления
  public virtual Color DefaultColor { get; set; }
  public Model Model { get; set; }
}
```

Наиболее распространенные тела — объекты простой геометрической формы для удобства реализованы наследниками класса Body:

- Cylinder — усеченный цилиндр, дополнительно к свойствам родительского класса имеет радиусы оснований, высоту и свойства, определяющие окраску оснований.
- Box — прямоугольный параллелепипед, имеет линейные размеры и шесть свойств, определяющих окраску сторон.
- Ball — шар, имеет свойство «радиус».

Понятно, что использование только трех примитивных сущностей для моделирования любых более-менее сложных объектов недостаточно. Во-первых, непонятно, как определить конструкцию, которая содержит движущиеся части (например, робот с установленным на него манипулятором). Во-вторых, таким способом не эффективно определять сложные модели, состоящие из большого числа полигонов или имеющие нетривиальную форму.

Для решения первой проблемы реализована возможность создания деревьев тел. Класс Body содержит коллекцию Nested, которая содержит его детей, и свойство Parent, указывающее на «родительское» тело. Для поддержания корректной структуры дерева, коллекция Nested доступна только для чтения; структуру дерева можно изменять только при помощи методов Add (добавляет тело в коллекцию Nested), Remove (удаляет тело из коллекции) и Clear (очищает коллекцию). Методы Add и Remove могут работать как с листьями (телами, у которых нет детей), так и с поддеревьями тел; свойство Parent при этом изменяется автоматически. Древовидная структура достаточно интуитивна, удобна при использовании из C#-кода и легко отображается на XML (такую возможность планируется реализовать в будущем для того, чтобы моделируемый мир можно было задавать без использования IDE).

Для определения объектов, имеющих сложную форму, добавим свойство Model. Model — это объект, который может содержать 3D-модель данного тела; 3D-модель может состоять из полигональной сетки, материалов и текстур. В случае, если тело представлено одним из примитивных классов (Ball, Box, Cylinder), но при этом имеет непустую модель, то именно модель используется для отображения в представлении. Примитивная форма тела при этом используется при расчете физических взаимодействий (для уменьшения вычислительной сложности и общего упрощения системы). Таким образом обеспечивается графическое разнообразие, но сохраняется простая логическая структура модели (модели — в смысле MVC). При определении мира разработчик может пользоваться простыми примитивами, а может получать более сложное графическое представление при помощи внешних моделей.

```
public class Body:INotifyPropertyChanged
{
  /*
  ...
  */
  public event PropertyChangedEventHandler PropertyChanged;
  public event Action<Body> ChildAdded;
  public event Action<Body> ChildRemoved;
}
```

В соответствии с принципами MVC, модель не содержит ссылок на контроллер или представления. Оповещение об изменениях модели происходит при помощи событий. Body реализует интерфейс INotifyPropertyChanged и при изменении любых свойств поднимает событие PropertyChanged. Для оповещения об изменении коллекции вложенных тел используются события ChildAdded и ChildRemoved.

Основные достоинства рассмотренной архитектуры:

- Простота. Модель (мир соревнований) описывается с использованием очевидных свойства, таких, как линейные размеры, плотность, и т.д. Сложные конструкции можно определять при помощи естественного отношения вложенности.

- Независимость от реализации контроллера и представления. Тела не ссылаются на внешние библиотеки и не используют никаких сторонних сущностей (например, специфических для 3D-графики или физического движка). Это уменьшает связность кода, упрощает тестирование и позволяет при необходимости заменить одну реализацию представления на другую. Так, например, в качестве альтернативы описанного в данной работе представления, основанного на DirectX, было реализовано (пока экспериментальное) WebGL-представление.

Литература

1. CVARC: an educational project for a gentle introduction to autonomous robots' control / Y. Okulovsky, P. Abduramanov, M. Kropotov, A. Ryabykh. — 2013. — Статья будет представлена на конференции Robotics in Education в сентябре 2013 в городе Лодзь, Польша.

2. Microsoft Developer Network. Model-View-Controller. — `http://msdn.microsoft.com/en-us/library/ff649643.aspx`.

3. Koenig N., Howard A. Design and use paradigms for Gazebo, an open-source multi-robot simulator // In IEEE/RSJ International Conference on Intelligent Robots and Systems. — 2004. — P. 2149–2154.

4. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo / Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher et al. // Simulation, Modeling, and Programming for Autonomous Robots. — Springer Berlin Heidelberg, 2012. — Vol. 7628 of Lecture Notes in Computer Science. — P. 400–411.

5. Google AI Challenge Tools source code. — `http://aichallenge.org/tools.tar.bz2`.

6. USARSim: a robot simulator for research and education / S. Carpin, M. Lewis, Jijun Wang et al. // Robotics and Automation, 2007 IEEE International Con¨ference on. — 2007. — P. 1400–1405.

7. Рябых А.В., Кропотов М.С., Окуловский Ю.С. Робототехнический эмулятор Eurosim // Современные проблемы математики. Тезисы Международной (43-й Всероссийской) молодежной школы-конференции. — 2012. — С. 232–234.

8. Can I use WebGL. — http://caniuse.com/webgl.

9. Документация SlimDX. — http://slimdx.org/docs/.

10. Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Addison-Wesley Professional, 1994.

11. Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering 3rd Edition. — Natick, MA, USA : A. K. Peters, Ltd., 2008. — P. 1045.

12. Eurobot 2012 rules. — http://www.swisseurobot.ch/images/2012/e2012_rules_en_final.pdf.

13. RuCTF. — http://www.ructf.org.

14. ImageCLEF — Image Retreival in CLEF. — http://www.imageclef.org/2012/robot.

15. M. Reckhaus, N. Hochgeschwender, J. Paulus. An overview of simulation and emulation in robotics // Simulation, Modeling, and Programming for Autonomous Robots. Second International Conference Proceedings. — SIMPAR '10. — Berlin, Heidelberg : Springer-Verlag, 2010. — P. 365–374.

16. Siegwart R., Nourbakhsh I. R. Introduction to Autonomous Mobile Robots. — Scituate, MA, USA : Bradford Company, 2004.