ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

Согласовано

Доцент департамента
программной инженерии
факультета компьютерных наук
канд. техн. наук

______________ Ахметсафина Р. З.

”____”_________ 2016 г

Утверждаю

Академический руководитель
образовательной программы
«Программная инженерия»
профессор департамента программной
инженерии канд. техн. наук

______________ Шилов В. В.

”____”_________ 2016 г

ПРОГРАММА СКЕЛЕТНАЯ АНИМАЦИЯ

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.509000 12 01-1

Студент группы БПИ 151 НИУ ВШЭ
______________ Абрамов А.М.
”____”_________ 2016 г

2016

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

УТВЕРЖДЕНО
RU.17701729.509000 12 01-1

ПРОГРАММА СКЕЛЕТНАЯ АНИМАЦИЯ

Текст программы

RU.17701729.509000 12 01-1

Листов 67

2016

Инв. № подл.	
Подп. и дата	
Взам. инв. №	
Инв. № дубл.	
Подп. и дата	

Содержание

1. Текст программы

1.1. ActionState

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.IO;        // for MemoryStream
using System.Reflection;
using System.Diagnostics;
using Assimp;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using Assimp.Configs;
using d2d = System.Drawing.Drawing2D;
using tk = OpenTK;
using Matrix4 = OpenTK.Matrix4;

namespace WinFormAnimation2D
{

    /// <summary>
    /// This class knows what argumets to pass to NodeInterpolator.
    /// </summary>
    class ActionState : BaseForEventDriven
    {

        public Animation _action;

        // owner = only used to get the global transform matrix for root bone
        public Entity _owner;
        public Matrix4 GlobalTransform
        {
            get {
                Debug.Assert(_owner != null);
                return _owner._transform._matrix;
            }
        }

        // index of keyframe maps to its time in ticks
        public List<double> KeyframeTimes;
        public int KeyframeCount
        {
            get { return KeyframeTimes.Count; }
        }
        public int FinalKeyframe
        {
            get { return KeyframeCount - 1; }
        }

        public string Name
        {
            get { return _action.Name; }
        }
        /// Duration of animation.
        public double TotalDurationSeconds
        {
            get { return _action.DurationInTicks * _action.TicksPerSecond; }
        }
        public double TotalDurationTicks
        {
            get { return _action.DurationInTicks; }
        }

        /// position of the time cursor in ticks of animation.
        public double TimeCursorInTicks
        {
            get
            {
                double interval_ticks = (KeyframeTimes[TargetKeyframe] - KeyframeTimes[OriginKeyframe]);
                return KeyframeTimes[OriginKeyframe] + interval_ticks * KfBlend;
            }
```

```
}

public double IntervalLengthMilliseconds
{
    get
    {
        double interval_ticks = Math.Abs(KeyframeTimes[TargetKeyframe] - KeyframeTimes[OriginKeyframe]);
        double interval_seconds = interval_ticks * _action.TicksPerSecond;
        return interval_seconds * 1000.0;
    }
}

/// TickPerSec can be used to change speed.
private double _tps;
public double TickPerSec
{
    get { return _tps; }
    set { _tps = value; }
}

/// Start or origin keyframe
private int _origin_keyframe;
public int OriginKeyframe
{
    get { return _origin_keyframe; }
    set
    {
        // Note: frame is strictly less than KeyframeCount
        if (0 <= value && value < KeyframeCount)
        {
             _origin_keyframe = value;
        }
    }
}

/// End or target keyframe
private int _target_keyframe;
public int TargetKeyframe
{
    get { return _target_keyframe; }
    set
    {
        // Note: frame is strictly less than KeyframeCount
        if (0 <= value && value < KeyframeCount)
        {
             _target_keyframe = value;
        }
    }
}

/// Blend value between 0.0 - 1.0, how much in between two keyframes are we
private double _kf_blend;
public double KfBlend
{
    get { return _kf_blend; }
    set
    {
        _kf_blend = Math.Min(Math.Max(0, value), 1.0);
        NotifyPropertyChanged();
    }
}

/// Automatically play the animation again after it has timed out.
public bool _loop;
public bool Loop
{
    get {
        return _loop;
    }
    set {
        _loop = value;
        if (_loop)
        {
            SetTime(0);
        }
        NotifyPropertyChanged();
    }
}

public ActionState(Animation action)
{
    SetCurrentAction(action);
```

```
        }

        public void NextInterval()
        {
            OriginKeyframe = Loop ? TargetKeyframe % (FinalKeyframe) : TargetKeyframe;
            TargetKeyframe = OriginKeyframe + 1;
            KfBlend = 0.0;
        }

        public void ReverseInterval()
        {
            OriginKeyframe = TargetKeyframe;
            TargetKeyframe -= 1;
            KfBlend = 1.0 - KfBlend;
        }

        /// Change the animation track. If there is more than one. We don't support this yet.
        public void SetCurrentAction(Animation action)
        {
            _action = action;
            _tps = action.TicksPerSecond;
            KfBlend = 0;
            // Keyframe times must be initialised before Origin/Target Keyframes
            KeyframeTimes = _action.NodeAnimationChannels[0].PositionKeys.Select(vk => vk.Time).ToList();
            OriginKeyframe = 0;
            TargetKeyframe = 0;
        }

        public int FindStartFrameAtTime(double time_ticks)
        {
            Debug.Assert(time_ticks >= 0);
            // sometimes first time is non zero (e.g. 0.045)
            if (time_ticks <= KeyframeTimes[0])
            {
                return 0;
            }
            for (int i = 1; i < KeyframeCount; i++)
            {
                if (time_ticks < KeyframeTimes[i])
                {
                    return i - 1;
                }
            }
            // return last frame if not found (because of numerical inaccuracies?)
            return KeyframeCount - 1;
        }

        /// Set the current time for the animation.
        /// Note: all the calculations here are done in ticks.
        public void SetTime(double time_seconds)
        {
            double time_ticks = time_seconds * TickPerSec;
            // when time overflows we loop by default
            double time = time_ticks % TotalDurationTicks;
            int start_frame = FindStartFrameAtTime(time_seconds);
            int end_frame = (start_frame + 1) % KeyframeCount;
            double delta_ticks = KeyframeTimes[end_frame] - KeyframeTimes[start_frame];
            // when we looped the animation
            if (delta_ticks < 0.0)
            {
                delta_ticks += TotalDurationTicks;
            }
            double blend = (time - KeyframeTimes[start_frame]) / delta_ticks;
            // assign results
            OriginKeyframe = start_frame;
            TargetKeyframe = end_frame;
            KfBlend = blend;
        }

    }
}
```

1.2. BoneNode.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.IO;        // for MemoryStream
using System.Reflection;
using System.Diagnostics;
using Assimp;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using Assimp.Configs;
using d2d = System.Drawing.Drawing2D;
using tk = OpenTK;
using Matrix4 = OpenTK.Matrix4;

namespace WinFormAnimation2D
{

    // Node with extended properties
    class BoneNode
    {
        public Node _inner;
        public Matrix4 GlobalTransform;
        public Matrix4x4 GlobTrans
        {
            get { return GlobalTransform.eToAssimp(); }
            set { GlobalTransform = value.eToOpenTK(); }
        }
        public Matrix4 LocalTransform;
        public Matrix4x4 LocTrans
        {
            get { return LocalTransform.eToAssimp(); }
            set {  LocalTransform = value.eToOpenTK(); }
        }

        public BoneNode Parent;
        public List<BoneNode> Children;

        public BoneNode(Node assimp_node)
        {
            _inner = assimp_node;
            Children = new List<BoneNode>(assimp_node.ChildCount);
        }

    }
```

1.3. MatrixExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ai = Assimp;
using Assimp.Configs;
using System.Windows.Forms;
using System.Drawing;
using d2d = System.Drawing.Drawing2D;
using tk = OpenTK;

namespace WinFormAnimation2D
{
    static class AssimpMatrixExtensions
    {

        /// <summary>
        /// Transform a direction vector by the given Matrix. Note: this is for assimp
        /// matrix which is row major.
        /// </summary>
        /// <param name="vec">The vector to transform</param>
        /// <param name="mat">The desired transformation</param>
        /// <param name="result">The transformed vector</param>
        public static ai.Vector3D eTransformVector(this ai.Matrix4x4 mat, ai.Vector3D vec)
        {
            return new ai.Vector3D
            {
                X = vec.X * mat.A1
                    + vec.Y * mat.B1
                    + vec.Z * mat.C1
                    + mat.A4,
                Y = vec.X * mat.A2
                    + vec.Y * mat.B2
                    + vec.Z * mat.C2
                    + mat.B4,
                Z = vec.X * mat.A3
                    + vec.Y * mat.B3
                    + vec.Z * mat.C3
                    + mat.C4
            };
        }

        /// <summary>
        /// Convert 4x4 Assimp matrix to OpenTK matrix.
        /// Will be a very useful function becasue Assimp
        /// matrices are very limited.
        /// </summary>
        /// <param name="m"></param>
        /// <returns></returns>
        public static tk.Matrix4 eToOpenTK(this ai.Matrix4x4 m)
        {
            return new tk.Matrix4
            {
                M11 = m.A1,
                M12 = m.B1,
                M13 = m.C1,
                M14 = m.D1,
                M21 = m.A2,
                M22 = m.B2,
                M23 = m.C2,
                M24 = m.D2,
                M31 = m.A3,
                M32 = m.B3,
                M33 = m.C3,
                M34 = m.D3,
                M41 = m.A4,
                M42 = m.B4,
                M43 = m.C4,
                M44 = m.D4
            };
        }

        /// <summary>
        /// Convert assimp 4 by 4 matrix into 3 by 2 matrix from System.Drawing.Drawing2D and use it
        /// for drawing with Graphics object.
        /// </summary>
        public static d2d.Matrix eTo3x2(this ai.Matrix4x4 m)
        {
            return new d2d.Matrix(m.A1, m.B1, m.A2, m.B2, m.A4, m.B4);
            // return new draw2D.Matrix(m[0, 0], m[1, 0], m[0, 1], m[1, 1], m[0, 3], m[1, 3]);
        }

        public static ai.Matrix4x4 eSnapTranslation(this ai.Matrix4x4 m, ai.Vector3D vec)
        {
            throw new NotImplementedException("Either make this method for assimp use, or change to OpenTK matrices!");
        }

        public static ai.Vector3D eGetTranslation(this ai.Matrix4x4 m)
        {
            return new ai.Vector3D(m.A4, m.B4, m.C4);
        }

    }
}
```

1.4. QuaternionExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.IO;        // for MemoryStream
using System.Reflection;
using System.Diagnostics;
using Assimp;
using Assimp.Configs;
using d2d = System.Drawing.Drawing2D;
using tk = OpenTK;

namespace WinFormAnimation2D
{
    static class AssimpQuaternionExtensions
    {
        public static Matrix4x4 eToMatrix(this Quaternion q)
        {
            float w = q.W, x = q.X, y = q.Y, z = q.Z;
            float xx = 2.0f * x * x;
            float yy = 2.0f * y * y;
            float zz = 2.0f * z * z;
            float xy = 2.0f * x * y;
            float zw = 2.0f * z * w;
            float xz = 2.0f * x * z;
            float yw = 2.0f * y * w;
            float yz = 2.0f * y * z;
            float xw = 2.0f * x * w;
            return new Matrix4x4(1.0f-yy-zz, xy + zw, xz - yw, 0.0f,
                                 xy - zw, 1.0f-xx-zz, yz + xw, 0.0f,
                                 xz + yw, yz - xw, 1.0f-xx-yy, 0.0f,
                                 0.0f, 0.0f, 0.0f, 1.0f);
        }

        public static tk.Quaternion eToOpenTK(this Quaternion q)
        {
            return new tk.Quaternion(q.X, q.Y, q.Z, q.W);
        }
    }
}
```

1.5. VectorExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ai = Assimp;
using Assimp.Configs;
using System.Windows.Forms;
using System.Drawing;
using d2d = System.Drawing.Drawing2D;
using tk = OpenTK;

namespace WinFormAnimation2D
{
    static class AssimpVectorExtensions
    {

        /// <summary>
        /// Convert assimp 3D vector to 2D System.Drawing.Point
        /// for drawing with Graphics object.
        /// </summary>
        public static Point eToPoint(this ai.Vector3D v)
        {
            return new Point((int)v.X, (int)v.Y);
        }

        /// <summary>
        /// Convert assimp 3D vector to 2D System.Drawing.PointF (floating point)
        /// for drawing with Graphics object.
        /// </summary>
        public static PointF eToPointFloat(this ai.Vector3D v)
        {
            return new PointF(v.X, v.Y);
        }

        /// <summary>
        /// Convert assimp 3D vector to opentk 2D vector.
        /// </summary>
        public static tk.Vector2 eAs2D_OpenTK(this ai.Vector3D v)
        {
            return new tk.Vector2(v.X, v.Y);
        }

        /// <summary>
        /// Convert assimp 3D vector to opentk 3D vector.
        /// </summary>
        public static tk.Vector3 eToOpenTK(this ai.Vector3D v)
        {
            return new tk.Vector3(v.X, v.Y, v.Z);
        }

    }
}
```

1.6. CameraDevice.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK;
using System.Drawing.Drawing2D;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;
using System.Runtime.CompilerServices;
using System.Diagnostics;

namespace WinFormAnimation2D
{

    enum CamMode
    {
        FreeFly
        , Orbital
    }

    /// <summary>
    /// Maintains camera abstraction. Allows support for orbiting, free fly and even 2D camera.
    /// </summary>
    class CameraDevice
    {
        /// Return the currently active camera mode.
        public CamMode _cam_mode
        {
            get { return Properties.Settings.Default.OrbitingCamera ? CamMode.Orbital : CamMode.FreeFly; }
        }
        public CameraFreeFly3D _3d_freefly;
        public OrbitCameraController _3d_orbital;

        /// Get the translation part of the camera matrix.
        public Vector3 GetTranslation
        {
            get
            { return (_cam_mode == CamMode.Orbital)
                    ? _3d_orbital.GetTranslation
                    : _3d_freefly.GetTranslation;
            }
        }

        /// Get the mouse position and calculate the world coordinates based on the screen coordinates.
        public Vector3 ConvertScreen2WorldCoordinates(Point screen_coords)
        {
            return Vector3.Zero;
        }

        /// Constructor
        public CameraDevice(Matrix4 opengl_init_mat)
        {
            _3d_freefly = new CameraFreeFly3D(opengl_init_mat);
            _3d_orbital = new OrbitCameraController();
        }
        /// Get the camera matrix to be uploaded to drawing 2D
        public Matrix4 MatrixToOpenGL()
        {
            return _cam_mode == CamMode.Orbital
                    ? _3d_orbital.MatrixToOpenGL()
                    : _3d_freefly.MatrixToOpenGL();
        }

        public void RotateAround(Vector3 axis)
        {
            _3d_freefly.ClockwiseRotateAroundAxis(axis);
            _3d_orbital.MouseMove((int)axis.X, (int)axis.Y);
            _3d_orbital.Scroll(axis.Z);
        }

        /// Respond to mouse events
        public void OnMouseMove(int x, int y)
        {
            _3d_freefly.ProcessMouse(x, y);
            _3d_orbital.MouseMove(x, y);
        }

        /// Zoom in/out of the scene.
        public void Scroll(float scroll)
        {
            _3d_freefly.MoveBy(new Vector3(0, 0, -1 * scroll));
            _3d_orbital.Scroll(scroll);
        }

        // x,y are direction parameters one of {-1, 0, 1}
        public void MoveBy(Vector3 direction)
        {
            _3d_freefly.MoveBy(direction);
            _3d_orbital.Pan(direction.X, direction.Y);
        }

    }

}
```

1.7. DrawConfig.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace WinFormAnimation2D
{
    /// This class will be passed into the Entity GetSettings() function to make the scene look best.
    class DrawConfig
    {
        // OpenGL settings
        // here is a template:
        /// Enable and disable OpenGL functionallity
        public bool EnableTexture2D = false;
        /// Enable and disable OpenGL functionallity
        public bool EnablePerspectiveCorrectionHint = false;
        /// Enable and disable OpenGL functionallity
        public bool EnableDepthTest = false;
        /// Enable and disable OpenGL functionallity
        public bool EnableFaceCounterClockwise = false;
        /// Enable and disable OpenGL functionallity
        public bool EnableDisplayList = false;
        /// Enable and disable OpenGL functionallity
        public bool EnablePolygonModeFill = false;
        /// Enable and disable OpenGL functionallity
        public bool EnablePolygonModeLine = false;
        /// Enable and disable OpenGL functionallity
        public bool EnableLight = false;

        public bool RenderWireframe = false;
        public bool RenderTextured = true;
        public bool RenderLit = true;
        public Pen DefaultPen = Pens.Gold;
        public Brush DefaultBrush = Brushes.Gold;

        // Font to be used for textual overlays in 3D view (size ~ 12px)
        public readonly Font DefaultFont12;
        // Font to be used for textual overlays in 3D view (size ~ 16px)
        public readonly Font DefaultFont16;

        public DrawConfig()
        {
            DefaultFont12 = new Font(FontFamily.GenericSansSerif, 12);
            DefaultFont16 = new Font(FontFamily.GenericSansSerif, 16);
        }
    }

}
```

1.8. Entity.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Assimp;
using Assimp.Configs;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.IO;        // for MemoryStream
using System.Reflection;
using OpenTK;
using OpenTK.Graphics.OpenGL;
using System.Diagnostics;
using Quaternion = Assimp.Quaternion;

namespace WinFormAnimation2D
{

    /// <summary>
    /// Represents the currently loaded object.
    /// One day we will have lots of these.
    /// </summary>
    class Entity
    {
        public ActionState _action;
        public BoneNode _armature;
        public Node _node;
        public SceneWrapper _scene;
        public Geometry _extra_geometry;
        public DrawConfig _draw_conf;
        public TransformState _transform;
        public Dictionary<int,MeshDraw> _mesh_id2mesh_draw = new Dictionary<int,MeshDraw>();
        public Matrix4 Matrix
        {
            get { return _transform._matrix; }
            set { _transform._matrix = value; }
        }

        public string Name
        {
            get { return _node.Name; }
            set { _node.Name = value; }
        }
        public Vector2 GetTranslation
        {
            get { return Matrix.ExtractTranslation().eTo2D(); }
        }

        // the only public constructor
        // TODO: change the "Node mesh". This should point to MeshDraw object which is unique to each entity.
        public Entity(SceneWrapper sc, Node mesh, BoneNode armature, ActionState state)
        {
            _scene = sc;
            _node = mesh;
            _extra_geometry = new Geometry(sc._inner.Meshes, mesh, armature);
            _armature = armature;
            _action = state;
            _transform = new TransformState(Matrix4.Identity, 10, 17);
        }

        public void UploadMeshVBO(IList<Material> materials)
        {
            InnerMakeMeshDraw(_scene._inner.Meshes, materials);
        }

        // Make a class that will be responsible for managind the buffer lists
        public void InnerMakeMeshDraw(IList<Mesh> meshes, IList<Material> materials)
        {
            for (int i = 0; i < meshes.Count; i++)
            {
                _mesh_id2mesh_draw[i] = new MeshDraw(meshes[i], materials);
            }
        }

        public void RotateBy(double angle_degrees)
        {
            _transform.Rotate(angle_degrees);
        }

        // x,y are direction parameters one of {-1, 0, 1}
        public void MoveBy(int x, int y)
        {
            var translate = _transform.TranslationFromDirection(new Vector3(x, y, 0));
            _transform.ApplyTranslation(translate);
        }

        public bool ContainsPoint(Vector2 p)
        {
            // modify the point so it is in entity space
            Vector3 tmp = new Vector3(p.X, p.Y, 0.0f);
            return _extra_geometry.EntityBorderContainsPoint(tmp.eTo2D());
        }

        /// Render the model stored in EntityScene useing the DrawConfig settings object.
        public void RenderModel(DrawConfig settings)
        {
            _draw_conf = settings;
            if (_draw_conf.EnablePerspectiveCorrectionHint)
            {
                // all are from System.Drawing.Drawing2D.
            }
            // second pass: render with this matrix
            RecursiveRenderSystemDrawing(_node);
            // apply the matrix to graphics just to draw the rectangle
            // TODO: we should just transform the border according to the RecursiveTransformVertices
            RenderBoundingBoxes(_extra_geometry);
        }

        // Render the scene.
        // each vertex at most one bone policy
        private void RecursiveRenderSystemDrawing(Node nd)
        {
            foreach(int mesh_id in nd.MeshIndices)
            {
                MeshDraw mesh_draw = _mesh_id2mesh_draw[mesh_id];
                mesh_draw.RenderVBO();
            }
            foreach (Node child in nd.Children)
            {
                RecursiveRenderSystemDrawing(child);
            }
        }

        public void RenderBoundingBoxes(Geometry geom)
        {
            foreach (var aabb in geom._mesh_id2box.Values)
            {
                if (Properties.Settings.Default.RenderAllMeshBounds)
                {
                    aabb.Render();
                }
            }
        }

        /// Deform the model vertices to align with the skeleton.
        public void UpdateModel(double dt_ms)
        {
            // first pass: calculate a matrix for each vertex
            RecursiveCalculateVertexTransform(_node, Matrix4.Identity.eToAssimp());
            RecursiveTransformVertices(_node);
        }

        // First pass: calculate the transofmration matrix for each vertex
        // here we must associate a matrix with each bone (maybe with each vertex_id??)
        // then we multiply the current_bone matrix with the one we had before
        // (perhaps it was identity, perhaps it was already some matrix (if
        // the bone influences many vertices) )
        // then we store this multiplied matrix.
        // in the render function we get a vertex_id, so we can find the matrix to apply
        // to the vertex, then we send the vertex to OpenGL
        /// Find the appropriate matrix to apply to the given vertex.
        public void RecursiveCalculateVertexTransform(Node nd, Matrix4x4 current)
        {
            Matrix4x4 current_node = current * nd.Transform;
            foreach(int mesh_id in nd.MeshIndices)
            {
                Mesh cur_mesh = _scene._inner.Meshes[mesh_id];
                MeshDraw mesh_draw = _mesh_id2mesh_draw[mesh_id];
                foreach (Bone bone in cur_mesh.Bones)
                {
                    // a bone transform is more than by what we need to trasnform the model
                    BoneNode armature_node = _scene.GetBoneNode(bone.Name);
                    Matrix4x4 bone_global_mat = armature_node.GlobTrans;
                    // bind tells the original delta in global coord, so we can find current delta
                    Matrix4x4 bind = bone.OffsetMatrix;
                    Matrix4x4 delta_roto = bind * bone_global_mat;
                    Matrix4x4 current_bone = delta_roto * current_node;
                    foreach (var pair in bone.VertexWeights)
                    {
                        // Can apply bone weight here
                        mesh_draw._vertex_id2matrix[pair.VertexID] = current_bone;
                    }
                }
            }
            foreach (Node child in nd.Children)
            {
                 RecursiveCalculateVertexTransform(child, current_node);
            }
        }

        /// <summary>Transform a Position by the given Matrix.
        /// Based on openTK compatiability vector 3 class.
        /// </summary>
        /// <param name="pos">The position to transform</param>
        /// <param name="mat">The desired transformation</param>
        /// <param name="result">The transformed position</param>
        public static void TransformPositionAssimp(ref Vector3D pos, ref Matrix4x4 mat, out Vector3D result)
        {
        // this is taken from https://github.com/opentk/opentk/blob/32665ca1cbdccb1c3be109ed0b7ff3f7cb5cb5b7/Source/Compatibility/Math/Vector3.c
            // Note that assimp is row major, while opentk is column major
            result.X = pos.X * mat.A1 +
                       pos.Y * mat.A2 +
                       pos.Z * mat.A3 +
                       mat.A4;

            result.Y = pos.X * mat.B1 +
                       pos.Y * mat.B2 +
                       pos.Z * mat.B3 +
                       mat.B4;

            result.Z = pos.X * mat.C1 +
                       pos.Y * mat.C2 +
                       pos.Z * mat.C3 +
                       mat.C4;
        }

        // Second pass: transform all vertices in a mesh according to bone
        // just apply the previously caluclated matrix
        public void RecursiveTransformVertices(Node nd)
        {
            foreach (int mesh_id in nd.MeshIndices)
            {
                MeshDraw mesh_draw = _mesh_id2mesh_draw[mesh_id];
                // map data from VBO
                IntPtr data;
                int qty_vertices;
                mesh_draw.BeginModifyVertexData(out data, out qty_vertices);
                // iterate over inital vertex positions
                Mesh cur_mesh = _scene._inner.Meshes[mesh_id];
                MeshBounds aabb = _extra_geometry._mesh_id2box[mesh_id];
```

```
                // go over every vertex in the mesh
                unsafe
                {
                    // array of floats: X,Y,Z.....
                    int sz = 3; // size of step
                    float* coords = (float*)data;
                    for (int vertex_id = 0; vertex_id < qty_vertices; vertex_id++)
                    {
                        Matrix4x4 matrix_with_offset = mesh_draw._vertex_id2matrix[vertex_id];
                        // get the initial position of vertex when scene was loaded
                        Vector3D vertex_default = cur_mesh.Vertices[vertex_id];
                        Vector3D vertex;
                        Entity.TransformPositionAssimp(ref vertex_default, ref matrix_with_offset, out vertex);
                        // write new coords back into array
                        coords[vertex_id*sz + 0] = vertex.X;
                        coords[vertex_id*sz + 1] = vertex.Y;
                        coords[vertex_id*sz + 2] = vertex.Z;
                    }
                }
                mesh_draw.EndModifyVertexData();

                foreach (Node child in nd.Children)
                {
                    RecursiveTransformVertices(child);
                }
            }
        }

    } // end of class

}
```

1.9. Bone.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Assimp;
using Assimp.Configs;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.IO;        // for MemoryStream
using System.Reflection;
using OpenTK;
using OpenTK.Graphics.OpenGL;
using System.Diagnostics;
using Quaternion = Assimp.Quaternion;

namespace WinFormAnimation2D
{
    struct BoundingVectors
    {
        public Vector3 ZeroNear;
        public Vector3 ZeroFar;
        public BoundingVectors(Vector3 near, Vector3 far)
        {
            ZeroNear = near;
            ZeroFar = far;
        }
    }

    class BoneBounds
    {
        public Vector3 _start;
        public Vector3 _end;

        // arbitrary vector that is perpendicular to the _end - _start
        // in 3D this might work better Vector3(-1*(_end.Y + _end.Z), 1, 1)
        // while in 2D use this Vector3(-1 * _end.Y, 1, 0), so that Z = 0;
        public Vector3 _normal
        {
            get {
                var bone_vec = _end - _start;
                var len = bone_vec.LengthFast;
            var sidevec = new Vector3(-1*(bone_vec.Y + bone_vec.X), 1.0f, 1.0f);
            return Vector3.Multiply(Vector3.NormalizeFast(sidevec), len/5.0f);
        }
    }

    public BoneBounds()
    {
        _start = Vector3.Zero;
        _end = Vector3.Zero;
    }

    public BoneBounds(Vector3 start, Vector3 end)
    {
        _start = start;
        _end = end;
    }

    // change from the 3d model into 2d program space just discard Z coordinate
    public Vector3[] Triangle
    {
        get
        {
            return new Vector3[] {
                _start
                , _start + _normal
                , _end
                , _start - _normal
                , _start
            };
        }
    }

    public void Render(Pen p = null)
    {
        // Util.GR.DrawLines(p == null ? Pens.Aqua : p, tmp);
        GL.Enable(EnableCap.ColorMaterial);
        GL.Material(MaterialFace.FrontAndBack, MaterialParameter.AmbientAndDiffuse, Color.Aqua);
        GL.Color3(Color.Aqua);
        GL.LineWidth(3.0f);
        GL.Begin(BeginMode.LineLoop);
        foreach (Vector3 vec in Triangle)
        {
            GL.Vertex3(vec.X, vec.Y, vec.Z);
        }
        GL.End();
    }
}

/// Stores info on extra geometry of the entity, bones that is.
class Geometry
{

    public Dictionary<int,MeshBounds> _mesh_id2box = new Dictionary<int,MeshBounds>();
    /// Bone name matched up with the triangle to render.
    public Dictionary<string,BoneBounds> _bone_id2triangle = new Dictionary<string,BoneBounds>();
    public BoundingBoxGroup EntityBox;
    public double _average_bone_length;

    /// Build geometry data for node (usually use only for one of the children of scene.RootNode)
    public Geometry(IList<Mesh> scene_meshes, Node nd, BoneNode armature)
    {
        MakeBoundingBoxes(scene_meshes, nd);
        MakeBoundingTriangles(armature);
        _average_bone_length = FindAverageBoneLength(armature);
        UpdateBonePositions(armature);
        EntityBox = new BoundingBoxGroup(_mesh_id2box.Values);
    }



    /// For the length of final children bones. Just use average length.
    public double FindAverageBoneLength(BoneNode nd)
    {
        double len = 0;
        int qty = 0;
        InnerFindAverageLength(nd, ref len, ref qty);
        return len / qty;
    }

    public void InnerFindAverageLength(BoneNode nd, ref double total_length, ref int bones_count)
    {
        var triangle = _bone_id2triangle[nd._inner.Name];
        Vector3 bone_start = nd.GlobalTransform.ExtractTranslation();
        // dont analyse bones with no children
        if (nd.Children.Count > 0)
        {
            // this bone's end == the beginning of __any__ child bone
            Vector3 bone_end = nd.Children[0].GlobalTransform.ExtractTranslation();
            double len = (bone_start - bone_end).Length;
            total_length += len;
            bones_count++;
            foreach (var child_nd in nd.Children)
            {
                InnerFindAverageLength(child_nd, ref total_length, ref bones_count);
            }
        }
    }

    /// Snap the render positions of bones, to deformations in the skeleton.
    public void UpdateBonePositions(BoneNode nd)
    {
        var triangle = _bone_id2triangle[nd._inner.Name];
        Vector3 new_start = nd.GlobalTransform.ExtractTranslation();
        if (nd.Children.Count > 0)
        {
            // this bone's end == the beginning of __any__ child bone
            Vector3 new_end = nd.Children[0].GlobalTransform.ExtractTranslation();
            triangle._start = new_start;
            triangle._end = new_end;
            foreach (var child_nd in nd.Children)
            {
                UpdateBonePositions(child_nd);
            }
        }
        else
        {
            // this bone has no children, we don't know where it will end, so we guess.
            // strategy 1: just set a random sensible value for bone
            // strategy 2: get geometric center of the vertices that this bone acts on
            // we have to use the Y-unit vector instead of X because we defined Y_UP
            // in the collada.dae file, so all the matrices work such that direct unit vector is unit Y
            // strategy 3: choose the length of the smallest bone found
            var delta = Vector3.TransformVector(Vector3.UnitY, nd.GlobalTransform);
            Vector3 new_end = new_start + Vector3.Multiply(delta, (float)_average_bone_length);
            triangle._start = new_start;
            triangle._end = new_end;
        }
    }

    // make triangles to draw for each bone
    private void MakeBoundingTriangles(BoneNode nd)
    {
        _bone_id2triangle[nd._inner.Name] = new BoneBounds();
        for (int i = 0; i < nd._inner.ChildCount; i++)
        {
             MakeBoundingTriangles(nd.Children[i]);
        }
    }

    /// For each node calculate the bounding box.
    /// This is used to align the viewport nicely when the scene is imported.
    private void MakeBoundingBoxes(IList<Mesh> scene_meshes, Node node)
    {
        foreach (int index in node.MeshIndices)
        {
            Mesh mesh = scene_meshes[index];
            _mesh_id2box[index] = new MeshBounds();
        }
        for (int i = 0; i < node.ChildCount; i++)
        {
            MakeBoundingBoxes(scene_meshes, node.Children[i]);
        }
    }

    public MeshBounds IntersectWithMesh(Vector2 point)
    {
        foreach (MeshBounds border in _mesh_id2box.Values)
        {
            if (border.CheckContainsPoint(point))
            {
                return border;
            }
        }
        return null;
```

```
        }

        public bool EntityBorderContainsPoint(Vector2 point)
        {
            return EntityBox.OverallBox.CheckContainsPoint(point);
        }

    }


}
```

1.10. MainForm.cs

```
using Assimp;
using Assimp.Configs;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using OpenTK;
using OpenTK.Graphics.OpenGL;

namespace WinFormAnimation2D
{
    public partial class MainForm : Form
    {

        MouseState _mouse = new MouseState();

        private World _world;

        RecentFilesFolders Recent = new RecentFilesFolders();

        private Stopwatch _last_frame_sw = new Stopwatch();
        private double LastFrameDelay;

        private bool LoadOpenGLDone;

        // State of the camera currently. We can affect this with buttons.
        private GUIConfig _gui_conf = new GUIConfig();
        private CommandLine _cmd;

        private IHighlightableNode last_selected_node;

        private Entity _current;
        private Entity Current
        {
            get { return _world._enttity_one; }
            set {
                _current = value;
                _cmd._current = value;
            }
        }

        private int TrackBarTimeRange
        {
            get { return this.trackBar_time.Maximum - this.trackBar_time.Minimum; }
        }

        private KeyboardInput _kbd;

        // camera related stuff
        private CameraDevice _camera;

        public MainForm()
        {
            InitializeComponent();
            this.checkBox_OpenGLDrawAxis.Checked = Properties.Settings.Default.OpenGLDrawAxis;
            this.toolStripStatusLabel_AnimTime.Text = "";
            _kbd = new KeyboardInput();
            Matrix4 opengl_camera_init = Matrix4.LookAt(0, 50, 500, 0, 0, 0, 0, 1, 0).Inverted();
            _camera = new CameraDevice(opengl_camera_init);
            // manually register the mousewheel event handler.
            this.glControl1.MouseWheel += new MouseEventHandler(this.glControl1_MouseWheel);
            _world = new World();
            _cmd = new CommandLine(_world, this);
            Recent.CurrentlyOpenFilePathChanged
                += (new_filepath) => this.Text = "Current file: " + new_filepath;
            RefreshOpenRecentMenu();
        }

        /// <summary>
        /// Get the items to show in open recent menu
        /// </summary>
        private void RefreshOpenRecentMenu()
        {
            // just replace old menu item wth a new one to refresh it
            Recent.ReplaceOpenRecentMenu(this.recentToolStripMenuItem
                , filepath => OpenFileCollada(filepath)
            );
        }

        public void SetAnimTime(double val)
        {
            this.toolStripStatusLabel_AnimTime.Text = val.ToString("F4");
            // if the user is not working with the track bar
            if (! this.trackBar_time.Focused)
            {
                double factor = TrackBarTimeRange / Current._action.TotalDurationSeconds;
                int track_val = (int)(val * factor);
                this.trackBar_time.Value = track_val;
            }
        }

    }
}
```

2. Приложение 1. Терминология

2.1. Терминология

Корневая вершина (англ. root node) Самый верхний узел дерева.

Полигональная сетка (жарг. меш от англ. polygon mesh) Совокупность вершин, рёбер и граней, которые определяют форму многогранного объекта в трехмерной компьютерной графике и объёмном моделировании. Гранями являются треугольники.

Дерево Связный ациклический граф. Связность означает наличие путей между любой парой вершин, ацикличность — отсутствие циклов и то, что между парами вершин имеется только по одному пути.

Степень вершины Количество инцидентных ей (входящих/исходящих из нее) ребер.

Интерполяция, интерполирование анимации Способ нахождения промежуточных значений состояния анимации по имеющемуся дискретному набору известных значений.

Z-буферизация В компьютерной трёхмерной графике способ учёта удалённости элемента изображения. Представляет собой один из вариантов решения «проблемы видимости»

Z-конфликт (англ. Z–fighting) Если два объекта имеют близкую Z-координату, иногда, в зависимости от точки обзора, показывается то один, то другой, то оба полосатым узором.

OpenGL (Open Graphics Library) Спецификация, определяющая независимый от языка программирования платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику. На платформе Windows конкурирует с Direct3D.

Рендеринг (англ. rendering — «визуализация») Термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы.

Текстура Растровое изображение, накладываемое на поверхность полигональной модели для придания ей цвета, окраски или иллюзии рельефа. Приблизительно использование текстур можно легко представить как рисунок на поверхности скульптурного изображения.

3. Приложение 3. Список используемой литературы

3.1. Список используемой литературы

1. ГОСТ 19.102-77 Стадии разработки. //Единая система программной документации. -М.: ИПК Издательство стандартов, 2001.
2. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению // Единая система программной документации. -М.:ИПК Издательство стандартов, 2001.
3. ГОСТ 19.101-77 Виды программ и программных документов //Единая система программной документации. -М.: ИПК Издательство стандартов, 2.: 001.

Лист регистрации изменений

Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ докум.	Входящий № сопроводительного докум. и дата	Подпись	Дата
	измененных	замененных	новых	аннулированных					