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

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

2016

УТВЕРЖДЕНО
RU.17701729.509000 12 01-1

ПРОГРАММА СКЕЛЕТНАЯ АНИМАЦИЯ

Текст программы

RU.17701729.509000 12 01-1

Листов 67

2016

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

Содержание

1. Текст программы

```
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;
        }

    }
}

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);
        }

    }


    // our job is to update the skeleton.
    // Entities should look up its status (current node transforms) during their rendering
    // , to make sure they are syncronised in position/rotation.
    //
    // Node animator knows about an action. It can perform the action on a given armature.
    // So it does: Snap this particular armature to this particular pose
    class NodeInterpolator
    {
        // Animation is what blender calls "action"
        // It is a set of keyframes that describe some action
        public Animation _action;
        public SceneWrapper _scene;

        public  NodeInterpolator(SceneWrapper sc, Animation action)
        {
            _scene = sc;
            _action = action;
        }

        // Update this particular armature to this particular frame in action (to this particular keyframe)
        public void ApplyAnimation(BoneNode armature, ActionState st)
        {
            ChangeLocalFixedDataBlend(st);
            var root_node = armature;
            root_node.GlobalTransform = root_node.LocalTransform * st.GlobalTransform;
            foreach (var child in root_node.Children)
            {
                ReCalculateGlobalTransform(child);
            }
        }

        /// <summary>
        /// Function to blend from one keyframe to another.
        /// </summary>
        public void ChangeLocalFixedDataBlend(ActionState st)
        {
            Debug.Assert(0 <= st.KfBlend && st.KfBlend <= 1);
            foreach (NodeAnimationChannel channel in _action.NodeAnimationChannels)
            {
                BoneNode bone_nd = _scene.GetBoneNode(channel.NodeName);
                // now rotation
                tk.Quaternion target_roto = tk.Quaternion.Identity;
                if (channel.RotationKeyCount > st.TargetKeyframe)
                {
                    target_roto = channel.RotationKeys[st.TargetKeyframe].Value.eToOpenTK();
                }
                tk.Quaternion start_frame_roto = channel.RotationKeys[st.OriginKeyframe].Value.eToOpenTK();
                tk.Quaternion result_roto = tk.Quaternion.Slerp(start_frame_roto, target_roto, (float)st.KfBlend);
                // now translation
                tk.Vector3 target_trans = tk.Vector3.Zero;
                if (channel.PositionKeyCount > st.TargetKeyframe)
                {
                    target_trans = channel.PositionKeys[st.TargetKeyframe].Value.eToOpenTK();
                }
                tk.Vector3 cur_trans = channel.PositionKeys[st.OriginKeyframe].Value.eToOpenTK();
                tk.Vector3 result_trans = cur_trans + tk.Vector3.Multiply(target_trans - cur_trans, (float)st.KfBlend);
                // combine rotation and translation
                tk.Matrix4 result = tk.Matrix4.CreateFromQuaternion(result_roto);
                result.Row3.Xyz = result_trans;
                bone_nd.LocTrans = result.eToAssimp();
            }
        }

        // Updates global transforms by walking the hierarchy
        private void ReCalculateGlobalTransform(BoneNode nd)
        {
            nd.GlobalTransform = nd.LocalTransform * nd.Parent.GlobalTransform;
            foreach (var child in nd.Children)
            {
                ReCalculateGlobalTransform(child);
            }
        }

    }

}



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 System.Diagnostics;


namespace WinFormAnimation2D
{
    class ArmatureEntity
    {
        public BoneNode _armature;
        public Scene _scene;

        public ArmatureEntity(Scene sc, BoneNode arma)
        {
            _armature = arma;
            _scene = sc;
        }


        public void RenderBone()
        {
        }

        //-------------------------------------------------------------------------------------------------
        // Render the scene.
        // Begin at the root node of the imported data and traverse
        // the scenegraph by multiplying subsequent local transforms
        // together on OpenGL matrix stack.
        // one mesh, one bone policy
        private void RecursiveRenderSystemDrawing(Node nd)
        {
        }

    } // end of class

}
```

```
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);
        }

    }
}

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);
        }
    }
}

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);
        }

    }
}

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 System.ComponentModel;
using System.Runtime.CompilerServices;

namespace WinFormAnimation2D
{
    class BaseForEventDriven : INotifyPropertyChanged
    {

        // boiler-plate INotifyPropertyChanged
        public event PropertyChangedEventHandler PropertyChanged;
        protected virtual void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
        protected void NotifyPropertyChanged([CallerMemberName] string propertyName = "")
        {
            OnPropertyChanged(propertyName);
        }
        protected bool NotifyUpdateField<T>(ref T field, T value, [CallerMemberName] string propertyName = "")
        {
            if (EqualityComparer<T>.Default.Equals(field, value)) return false;
            field = value;
            OnPropertyChanged(propertyName);
            return true;
        }
        // end boiler-plate
    }

}

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);
        }

    }

}

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
{
    class CameraDrawing2D : ITransformState
    {
        // we need half the size of picture box
        public float rotate_offset_x;
        public float rotate_offset_y;

        public TransformState _transform;
        public TransformState Transform
        {
            get { return _transform; }
        }

        public Vector3 GetTranslation
        {
            get { return _transform.GetTranslation; }
        }

        public Vector2 GetTranslation2D
        {
            get { return _transform.GetTranslation2D; }
        }

        public Matrix4 CamMatrix
        {
            get { return _transform._matrix; }
        }

        public void ProcessMouse(int x, int y)
        {
            // when user pulls mouse to the right (x > 0) we perform a clockwise rotation.
            if (x != 0)
            {
                RotateBy(x * _transform.RotateSpeedDegrees);
            }
        }

        /// <summary>
        /// Get the mouse position and calculate the world coordinates based on the screen coordinates.
        /// </summary>
        public Vector2 ConvertScreen2WorldCoordinates(Point screen_coords)
        {
            Vector3 tmp = new Vector3(screen_coords.X, screen_coords.Y, 0.0f);
            tmp = Vector3.Transform(tmp, _transform._matrix);
            return new Vector2(tmp.X, tmp.Y);
        }

        public void RotateBy(double angle_degrees)
        {
            RotateAroundScreenCenter2D(angle_degrees);
        }

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

        public CameraDrawing2D(Matrix4 draw2d_init_mat, Size window_size)
        {
            _transform = new TransformState(draw2d_init_mat, 10.0, 1.5);
            rotate_offset_x = window_size.Width / 2.0f;
            rotate_offset_y = window_size.Height / 2.0f;
        }

        /// <summary>
        /// Get the camera matrix to be uploaded to drawing 2D
        /// </summary>
        public Matrix4 MatrixToDrawing2D()
        {
            Matrix4 cam_inverted = _transform._matrix;
            cam_inverted.Invert();
            return cam_inverted;
        }

        // when doing a rotation we want to perform it around the screen center.
        public void RotateAroundScreenCenter2D(double angle_degrees)
        {
            float angle_radians = (float)(angle_degrees * Math.PI / 180.0);
            // we would remove the translation in OpenGL because its screen center is at (0,0,0)
            // in 2D camera screen center is at (Width/2.0, Height/2.0)
            // so translate to screen center
            _transform._matrix = Matrix4.CreateTranslation(rotate_offset_x, rotate_offset_y, 0.0f) * _transform._matrix;
            _transform._matrix = Matrix4.CreateRotationZ(angle_radians) * _transform._matrix;
            // translate back
            _transform._matrix = Matrix4.CreateTranslation(-rotate_offset_x, -rotate_offset_y, 0.0f) * _transform._matrix;
        }

    }
}

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
{
    class CameraFreeFly3D : ITransformState
    {
        public TransformState _transform;
        public TransformState Transform
        {
            get { return _transform; }
        }

        public Vector3 GetTranslation
        {
            get { return _transform.GetTranslation; }
        }

        public Vector2 GetTranslation2D
        {
            get { return _transform.GetTranslation2D; }
        }

        public Matrix4 CamMatrix
        {
            get { return _transform._matrix; }
        }

        public CameraFreeFly3D(Matrix4 opengl_init_mat)
        {
            _transform = new TransformState(opengl_init_mat, 10, 1.5);
        }

        /// <summary>
        /// Get the camera matrix to be uploaded to drawing 2D
        /// </summary>
        public Matrix4 MatrixToOpenGL()
        {
            Matrix4 opengl_cam_inverted = _transform._matrix;
            opengl_cam_inverted.Invert();
            return opengl_cam_inverted;
        }

        /// <summary>
        /// Get the mouse position and calculate the world coordinates based on the screen coordinates.
        /// </summary>
        public PointF ConvertScreen2WorldCoordinates(PointF screen_coords)
        {
            Vector3 tmp = new Vector3(screen_coords.X, screen_coords.Y, 0.0f);
            tmp = Vector3.Transform(tmp, _transform._matrix);
            return new PointF(tmp.X, tmp.Y);
        }

        // Movement in ZY plane
        public void ProcessMouse(int x, int y)
        {
            // when user pulls mouse to the right (x > 0) we perform a clockwise rotation.
            if (x != 0)
            {
                _transform.RotateAroundAxis(x * _transform.RotateSpeedDegrees, Vector3.UnitY);
                return;
            }
            if (y != 0)
            {
                _transform.RotateAroundAxis(y * _transform.RotateSpeedDegrees, Vector3.UnitX);
                return;
            }
        }

        public void RotateBy(double direction)
        {
            ClockwiseRotateAroundAxis(Vector3.Multiply(Vector3.UnitX, (float)direction));
        }
        public void ClockwiseRotateAroundAxis(Vector3 axis)
        {
            _transform.RotateAroundAxis(_transform.RotateSpeedDegrees, axis);
        }

        // x,y,z are direction parameters one of {-1, 0, 1}
        public void MoveBy(Vector3 dir)
        {
            _transform.MoveBy(dir);
        }

    }
}

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
{
    public class OrbitCameraController
    {
        private Matrix4 _view;
        private Matrix4 _viewWithOffset;
        private float _cameraDistance;
        private Vector3 _right;
        private Vector3 _up;
        private Vector3 _front;

        public Vector3 GetTranslation
        {
            get { return _viewWithOffset.ExtractTranslation(); }
        }
```

```
private Vector3 _panVector;

private bool _dirty = true;

private float ZoomSpeed = 2.00105f;
private float MinimumCameraDistance = 0.1f;
/// <summary>
/// Rotation speed, in degrees per pixels
/// </summary>
private float RotationSpeed = 0.5f;
private float PanSpeed = 2.0f; // 0.004f;
private float InitialCameraDistance = 200.0f;

private Vector3 _pivot;


public OrbitCameraController()
{
    // _view = Matrix4.CreateFromAxisAngle(new Vector3(0.0f, 1.0f, 0.0f), 0.9f);
    _view = Matrix4.Identity;
    _viewWithOffset = Matrix4.Identity;
    _cameraDistance = InitialCameraDistance;
    _right = Vector3.UnitX;
    _up = Vector3.UnitY;
    _front = Vector3.UnitZ;
    SetOrbitOrConstrainedMode();
}

public Matrix4 MatrixToOpenGL()
{
    return this.GetView(); // this.GetView().Inverted();
}

public void SetPivot(Vector3 pivot)
{
    _pivot = pivot;
    _dirty = true;
}

public Matrix4 GetView()
{
    if (_dirty)
    {
          UpdateViewMatrix();
    }
    return _viewWithOffset;
}

public void MouseMove(int x, int y)
{
    if(x == 0 && y == 0)
    {
        return;
    }
    if (x != 0)
    {
        _view *= Matrix4.CreateFromAxisAngle(_up, (float)(x * RotationSpeed * Math.PI / 180.0));
    }
    if (y != 0)
    {
        _view *= Matrix4.CreateFromAxisAngle(_right, (float)(y * RotationSpeed * Math.PI / 180.0));
    }
    _dirty = true;
    SetOrbitOrConstrainedMode();
}

public void Scroll(float z)
{
    _cameraDistance *= (float)Math.Pow(ZoomSpeed, -z);
    _cameraDistance = Math.Max(_cameraDistance, MinimumCameraDistance);
    _dirty = true;
}

public void Pan(float x, float y)
{
    _panVector.X += x * PanSpeed;
    _panVector.Y += -y * PanSpeed;
    _dirty = true;
}

public void MovementKey(float x, float y, float z)
{
```

```
            // TODO switch to FPS camera at current position?
        }

        public Vector3 _local_x { get { return _view.Row0.Xyz; } }
        public Vector3 _local_y { get { return _view.Row1.Xyz; } }
        public Vector3 _local_z { get { return _view.Row2.Xyz; } }
        public Vector3 _local_trans { get { return _view.Row3.Xyz; } }

        private void UpdateViewMatrix()
        {
            // for othagonal matrices T^(-1) = T^(transposed), so here we are applying a global rotation
            _viewWithOffset = Matrix4.LookAt(_view.Column2.Xyz * _cameraDistance + _pivot, _pivot, _view.Column1.Xyz);
            _viewWithOffset *= Matrix4.CreateTranslation(_panVector);
            _dirty = false;
        }

        /// Switches the camera controller between the X,Z,Y and Orbit modes.
        public void SetOrbitOrConstrainedMode()
        {
            _dirty = true;
        }

    }
}

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;

namespace WinFormAnimation2D
{
    class CommandLine
    {

        public World _world;
        public Timer _timer;
        public MainForm _form;
        public Entity _current;
        //public ListBox _box_debug;

        public EventHandler StepInterval;
        public EventHandler StepAll;
        public EventHandler DynamicTimeBlend;

        public bool NeedWindowRedraw;

        private IList<MethodInfo> _commands_cached = null;
        public IEnumerable<MethodInfo> Commands
        {
            get
            {
                if (_commands_cached == null)
                {
                    _commands_cached = this.GetType()
                        .GetMethods(BindingFlags.Public | BindingFlags.Instance)
                        .Where(f => char.IsLower(f.Name[0])).ToList();
                }
                return _commands_cached;
            }
        }

        // time of animation frame that was just rendered and time right now
        public Stopwatch anim_frame_time = new Stopwatch();

        public Dictionary<string, string> _debug = new Dictionary<string, string>();

        public CommandLine(World world, MainForm form)
        {
            _world = world;
    _timer = new Timer();
    _timer.Interval = 50;
    // _box_debug = debug;
    _form = form;
    StepInterval = delegate { this.stepf(); };
    StepAll = delegate { this.stepall(); };
    DynamicTimeBlend = delegate { this.DynamicStepTime(); };
}

public void ShowDebug()
{
    // this._box_debug.Items.Clear();
    // foreach (var v in _debug)
    // {
    //     _box_debug.Items.Add(v.Key + " = " + v.Value);
    // }
}

// jump to time directly
public void jumpt(double seconds)
{
    if (_current == null)
    {
        return;
    }
    _current._action.SetTime(seconds);
    _form.SetAnimTime(seconds);
    _world._action_one.ApplyAnimation(_current._armature, _current._action);
    NeedWindowRedraw = true;
}

// change the keyframe interval to go in reverse direction (back-play of animation)
public void bkf()
{
    if (_current == null)
    {
        return;
    }
    _current._action.ReverseInterval();
    _debug["from frame"] = _current._action.OriginKeyframe.ToString();
}

// change the keyframe interval to the next one
public void fkf()
{
    if (_current == null)
    {
        return;
    }
    _current._action.NextInterval();
    _debug["from frame"] = _current._action.OriginKeyframe.ToString();
}

// sets the blend value for current keyframe
public void blend(double percent)
{
    if (_current == null)
    {
        return;
    }
    _current._action.KfBlend = percent / 100.0;
    _debug["blend"] = _current._action.KfBlend.ToString();
}

// force applies animation to armature and causes a redraw
public void applyanim()
{
    if (_current == null)
    {
        return;
    }
    _world._action_one.ApplyAnimation(_current._armature, _current._action);
    NeedWindowRedraw = true;
}

// increment blend by time value proportional to delay from last frame
// automatically advance to next keyframe
public void DynamicStepTime()
{
    if (_current == null)
    {
        return;
    }
    double frame_millisecs = anim_frame_time.ElapsedMilliseconds;
    anim_frame_time.Restart();
    if (_current._action.KfBlend < 1.0)
    {
        double interval_millisecs = _current._action.IntervalLengthMilliseconds;
        // koefficient to map interval time into a 0..1 blend interval
        double k = 1.0 / interval_millisecs;
        // we know how much the time changed, now we need to find out how much to add to blend
        _current._action.KfBlend += (frame_millisecs * k);
    }
    else
    {
        _current._action.KfBlend = 0.0;
        _current._action.NextInterval();
    }
    _world._action_one.ApplyAnimation(_current._armature, _current._action);
    NeedWindowRedraw = true;
    _form.SetAnimTime(_current._action.TimeCursorInTicks);
}

// start the timer to play througth all keyframes with correct time
public void playall(bool on)
{
    if (_current == null)
    {
        return;
    }
    if (on)
    {
        anim_frame_time.Reset();
        anim_frame_time.Start();
        //_current._action.SetTime(0);
        _current._action.Loop = true;
        _timer.Tick += DynamicTimeBlend;
        if (_timer.Enabled == false)
        {
            _timer.Start();
        }
    }
    else
    {
        _current._action.Loop = false;
        _timer.Tick -=  DynamicTimeBlend;
    }
}

// start the timer to step in 0.1 blend size througth the current interval
// don't go to next keyframe
public void playinterval()
{
    if (_current == null)
    {
        return;
    }
    _timer.Tick += StepInterval;
    if (_timer.Enabled == false)
    {
        _timer.Start();
    }
}

// step through all animation with 0.1 blend interval
// basically a small jump forwards in time
public void stepall()
{
    if (_current == null)
    {
        return;
    }
    _current._action.SetTime(_current._action.TimeCursorInTicks + 0.8);
    //if (_current._action.KfBlend < 1.0)
    //{
    //    _current._action.KfBlend += 0.1;
    //}
    //else
    //{
    //    _current._action.KfBlend = 0.0;
    //    _current._action.NextInterval();
    //}
    _world._action_one.ApplyAnimation(_current._armature
        , _current._action);
    NeedWindowRedraw = true;
    _form.SetAnimTime(_current._action.TimeCursorInTicks);
}

// step in 0.1 blend interval, don’t overflow to next keyframe
public void stepf()
{
    if (_current == null)
    {
        return;
    }
    if (_current._action.KfBlend < 0.99)
    {
        _current._action.KfBlend += 0.1;
    }
    _world._action_one.ApplyAnimation(_current._armature
        , _current._action);
    NeedWindowRedraw = true;
    _form.SetAnimTime(_current._action.TimeCursorInTicks);
}

public void set(string name, string value)
{
    PropertyInfo[] possible = Properties.Settings.Default.GetType().GetProperties();
    PropertyInfo prop = possible.SingleOrDefault(p => p.Name == name);
    if (prop == null)
    {
        SetError("property to set not found");
        return;
    }
    // get converter for value
    var conv = TypeDescriptor.GetConverter(prop.PropertyType);
    // for converted output
    if (! conv.IsValid(value))
    {
        return;
    }
    prop.SetValue(Properties.Settings.Default, conv.ConvertFromString(value));
}

public void help()
{
    _debug["help"] = string.Join(", ", Commands.Select(f => f.Name));
    ShowDebug();
}

public void SetError(string msg)
{
    _debug["err"] = msg;
    ShowDebug();
}

public void RunCmd(string input)
{
    input = input.Trim(’ ’);
    IEnumerable<string> tokens;
    if (input.Contains(’ ’))
    {
        tokens = input.Split(’ ’);
    }
    else
    {
        tokens = new string[] { input };
    }
    int qty_args = tokens.Count() - 1;
    string fname = tokens.First();
    // find the function
    MethodInfo cmdinfo = Commands.SingleOrDefault(f => f.Name == (string)fname);
    if (cmdinfo == null)
    {
        SetError("command not found");
        help();
        return;
    }
    // get converter for each parameter
    IEnumerable<TypeConverter> arg_converters = cmdinfo.GetParameters()
        .Select(p => TypeDescriptor.GetConverter(p.ParameterType));
    if (qty_args < arg_converters.Count())
    {
        SetError("command takes " + arg_converters.Count() + " args");
        return;
    }
```

```
            // for converted output
            var fargs = new List<object>(qty_args);
            foreach (var pair in tokens.Skip(1).Zip(arg_converters, (token,conv) => new { t = token, c = conv}))
            {
                if (! pair.c.IsValid(pair.t))
                {
                    SetError("can not convert ’" + pair.t + "’");
                    return;
                }
                fargs.Add(pair.c.ConvertFromString(pair.t));
            }
            cmdinfo.Invoke(this, fargs.ToArray());
            ShowDebug();
        }
    }

}

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 Drawing2dGraphicsExtensions
    {

        /// <summary>
        /// Draw circle with Graphics from point and radius.
        /// </summary>
        public static void eDrawCircle(this Graphics g, Pen pen, Point p, int rad)
        {
            var rect = new RectangleF(p.X - rad, p.Y - rad, 2 * rad, 2 * rad);
            g.DrawEllipse(pen, rect);
        }

        /// <summary>
        /// Debug function to quickly draw points with Graphics
        /// </summary>
        public static void eDrawPoint(this Graphics g, Point p)
        {
            float rad = 0.3f;         // radius
            var rect = new RectangleF(p.X - rad, p.Y - rad, 2 * rad, 2 * rad);
            g.DrawEllipse(Util.pp3, rect);
        }

        /// <summary>
        /// Quick debug function to draw _FLOATING_ PointF with Graphics
        /// </summary>
        public static void eDrawPoint(this Graphics g, PointF p)
        {
            float rad = 0.03f;         // radius
            var rect = new RectangleF(p.X - rad, p.Y - rad, 2 * rad, 2 * rad);
            g.DrawEllipse(Util.pp3, rect);
        }

        /// <summary>
        /// Debug function to quickly draw _FLOATING_ points with Graphics
        /// </summary>
        public static void eDrawBigPoint(this Graphics g, PointF p)
        {
            float rad = 10.0f;         // radius
            var rect = new RectangleF(p.X - rad, p.Y - rad, 2 * rad, 2 * rad);
            g.DrawEllipse(Util.pp1, rect);
        }

    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
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;
using System.Diagnostics;

namespace WinFormAnimation2D
{
    static class Drawing2dMatrixExtensions
    {

        /// <summary>
        /// Transform a single PointF object and return the result.
        /// </summary>
        /// <param name="mat"></param>
        /// <param name="p"></param>
        /// <returns></returns>
        public static PointF eTransformSinglePointF(this d2d.Matrix mat, PointF p)
        {
            var tmp = new PointF[] { p };
            mat.TransformPoints(tmp);
            return tmp[0];
        }

        /// <summary>
        /// Transform a single Vector2 object and return the result.
        /// </summary>
        /// <param name="mat"></param>
        /// <param name="p"></param>
        /// <returns></returns>
        public static tk.Vector2 eTransformSingleVector2(this d2d.Matrix mat, tk.Vector2 p)
        {
            var tmp = new tk.Vector2[] { p };
            mat.eTransformVector2(tmp);
            return tmp[0];
        }

        /// <summary>
        /// Applies the geometric transform represented by this System.Drawing.Drawing2D.Matrix
        /// to a specified array of Opentk.Vector2
        /// </summary>
        /// <param name="mat"></param>
        /// <param name="vecs"></param>
        public static void eTransformVector2(this d2d.Matrix mat, tk.Vector2[] vecs)
        {
            PointF[] tmp = vecs.Select(vec => new PointF(vec.X, vec.Y)).ToArray();
            mat.TransformPoints(tmp);
            // set them equal this way we dont mess up if other
            // objects kept pointers to some vector and we just override it
            for (int i = 0; i < vecs.Length; i++)
            {
                vecs[i].X = tmp[i].X;
                vecs[i].Y = tmp[i].Y;
            }
        }

        /// <summary>
        /// Rescale the matrix. Preserve rotation and translation.
        /// </summary>
        /// <param name="mat"></param>
        /// <returns></returns>
        public static d2d.Matrix eSnapScale(this d2d.Matrix mat, double scale = 1.0)
        {
            var curmat = mat.Elements;
            // normalise the x and y axis to set scale to 1.0f
            var x_axis = new ai.Vector2D(curmat[0], curmat[1]);
            var y_axis = new ai.Vector2D(curmat[2], curmat[3]);
            x_axis.Normalize();
            y_axis.Normalize();
            // scale the axis
            x_axis.X *= (float)scale;
            x_axis.Y *= (float)scale;
            y_axis.X *= (float)scale;
            y_axis.Y *= (float)scale;
            // make new matrix with scale of 1.0f
            // Do not change the translation
            var newmat = new d2d.Matrix(x_axis[0], x_axis[1]
                , y_axis[0], y_axis[1]
                , curmat[4], curmat[5]
            );
            return newmat.Clone();
        }

        /// <summary>
        /// Snap translation part of the matrix to a given vector.
        /// </summary>
        /// <param name="mat"></param>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns></returns>
        public static d2d.Matrix eSnapTranslate(this d2d.Matrix mat, double x, double y)
        {
            var curmat = mat.Elements;
            var newmat = new d2d.Matrix(curmat[0], curmat[1]
                , curmat[2], curmat[3]
                , (float)x, (float)y);
            return newmat.Clone();
        }

        /// <summary>
        /// Snap rotate to some angle. Preserve scale and translation.
        /// </summary>
        /// <param name="mat"></param>
        /// <param name="angle">__ANGLE IS IN DEGREES__</param>
        /// <returns></returns>
        public static d2d.Matrix eSnapRotate(this d2d.Matrix mat, double angle)
        {
            // Graphics tries to work opposite of OpenGL, in Drawing2D:
            // PRE - multiply for local
            // post -multiply for global
            var curmat = mat.Elements;
            // get the vector components.
            var x_axis = new ai.Vector2D(curmat[0], curmat[1]);
            var y_axis = new ai.Vector2D(curmat[2], curmat[3]);
            // Get the scale of current matrix
            double x_len = x_axis.Length();
            double y_len = y_axis.Length();
            var newmat = new d2d.Matrix();
            // Preserve scale and translation
            // This means: v*M = v*(S * R * T)
            newmat.Scale((float)x_len, (float)y_len);
            newmat.Rotate((float)angle);
            newmat.Translate(curmat[4], curmat[5]);
            return newmat.Clone();
        }

        /// <summary>
        /// Returns the translation component of matrix as a Point
        /// </summary>
        /// <param name="mat"></param>
        /// <returns></returns>
        static public PointF eGetTranslationPoint(this d2d.Matrix mat)
        {
            return new PointF(mat.Elements[4], mat.Elements[5]);
        }
    }
}

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);
        }
    }


    class GUIConfig
    {
        // should animation be playing. This should really go into GUISettings
        private bool Animating = false;

        /// Show we render Frames Per Second counter?
        public bool ShowFps = true;

        /// Currently active scene
        public Entity CurrentEntity;

        /// Enum of all supported camera modes.
        public enum CameraMode
        {
            Fps = 0,
            Orbit,
            _Max
        }
        public CameraMode CamMode = CameraMode.Orbit;

        public GUIConfig()
        {
            // nothing to do here
        }
    }

}

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;

// TODO: this is piece of text taken from function that no longer exists.
// but it is trying to describe my architechture. But it is getting old and useless.
// HERE GOES:
// setup specific to this scene what other objects do not know about. (wireframe, texture,material,scale...)
// GetRenderSettings gets the currently active globale settings for the program.
// it looks at them and chooses the best settings for itself taking into
// consideration the globals. So it tries to get the scene looking ideal while
// still respecting global user settings (like: draw in wireframe, or without texture)
// that are currently turned on in the application. This settings are
// activated back in the DrawToOpenGL class. After their activation we call the
// render method on this particular object that pushes vertices (not settings) to OpenGL.
// This object should have some render code.

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

}

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();
        }
    }

    class MeshBounds
    {
        public Vector3 _zero_near;
        public Vector3 _zero_far;
        public bool _updating;

        public Vector3 Center
        {
            get { return Vector3.Divide(Vector3.Add(_zero_near,_zero_far), 2.0f);  }
        }

        // change from the 3d model into 2d program space just discard Z coordinate
        public RectangleF Rect
        {
            get
            {
                return new RectangleF(_zero_near.X, _zero_near.Y
                    , _zero_far.X - _zero_near.X
                    , _zero_far.Y - _zero_near.Y
                );
            }
        }

        public MeshBounds(Vector3D zero_near, Vector3D zero_far)
        {
            _zero_far = zero_far.eToOpenTK();
            _zero_near = zero_near.eToOpenTK();
        }

        public MeshBounds()
        {
            _zero_near = new Vector3(float.MaxValue, float.MaxValue, float.MaxValue);
            _zero_far = new Vector3(float.MinValue, float.MinValue, float.MinValue);
        }

        public bool CheckContainsPoint(Vector2 p)
        {
            if ((_zero_near.X < p.X && p.X < _zero_far.X)
                && (_zero_near.Y < p.Y && p.Y < _zero_far.Y))
            {
                return true;
            }
            return false;
        }

        public void Render()
        {
            RenderGL();
        }

        public void RenderGL()
        {
            GL.Color3(Util.cc4);
            GL.Normal3(0, 1, 1);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
            GL.Begin(BeginMode.Quads);
            GL.Vertex3(Rect.Location.X, Rect.Location.Y, 1.0);
            GL.Vertex3(Rect.Location.X + Rect.Width, Rect.Location.Y, 1.0);
            GL.Vertex3(Rect.Location.X + Rect.Width, Rect.Location.Y + Rect.Height, 0.0);
            GL.Vertex3(Rect.Location.X, Rect.Location.Y + Rect.Height, 0.0);
            GL.End();
        }

        public BoundingVectors GetNearFar()
        {
            return new BoundingVectors(_zero_near, _zero_far);
        }

        // call this before starting a cycle of updates
        public void SafeStartUpdateNearFar()
        {
            if (_updating)
            {
                return;
            }
            _zero_near = new Vector3(float.MaxValue, float.MaxValue, float.MaxValue);
            _zero_far = new Vector3(float.MinValue, float.MinValue, float.MinValue);
            _updating = true;
        }

        public void EndUpdateNearFar()
        {
            Debug.Assert(_updating == true, "Update was never started");
            _updating = false;
        }

        // pass in a vertex belonging to the mesh, we will
        // check if we need to change the near/far values
        public void UpdateNearFar(Vector3 vertex)
        {
            // update frame min
            _zero_near.X = Math.Min(_zero_near.X, vertex.X);
            _zero_near.Y = Math.Min(_zero_near.Y, vertex.Y);
            _zero_near.Z = Math.Min(_zero_near.Z, vertex.Z);
            // update frame max
            _zero_far.X = Math.Max(_zero_far.X, vertex.X);
            _zero_far.Y = Math.Max(_zero_far.Y, vertex.Y);
            _zero_far.Z = Math.Max(_zero_far.Z, vertex.Z);
        }

    }

    class BoundingBoxGroup
    {
        public List<MeshBounds> Items;
        public MeshBounds _overall_box = new MeshBounds();
        public MeshBounds OverallBox
        {
            get
            {
                // Update before returning
                var tmp = GetCoveringBoundingBox(Items);
                _overall_box._zero_near = tmp._zero_near;
                _overall_box._zero_far = tmp._zero_far;
                return _overall_box;
            }
        }

        public BoundingBoxGroup(IEnumerable<MeshBounds> boxes)
        {
            Items = boxes.ToList();
        }

        public MeshBounds GetCoveringBoundingBox(IEnumerable<MeshBounds> boxes)
        {
            Vector3D zero_near = new Vector3D(float.MaxValue, float.MaxValue, float.MaxValue);
            Vector3D zero_far = new Vector3D(float.MinValue, float.MinValue, float.MinValue);
            foreach (var aabb in boxes)
            {
                // find min
                zero_near.X = Math.Min(zero_near.X, aabb._zero_near.X);
                zero_near.Y = Math.Min(zero_near.Y, aabb._zero_near.Y);
                zero_near.Z = Math.Min(zero_near.Z, aabb._zero_near.Z);
                // find max
                zero_far.X = Math.Max(zero_far.X, aabb._zero_far.X);
                zero_far.Y = Math.Max(zero_far.Y, aabb._zero_far.Y);
                zero_far.Z = Math.Max(zero_far.Z, aabb._zero_far.Z);
            }
            return new MeshBounds(zero_near, zero_far);
        }
    }

    /// 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);
        }

    }


}

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;

namespace WinFormAnimation2D
{
    enum KeyboardAction
    {
            None
            , DoRotation
            , DoMotion
            , RunCommand
    }

    class KeyboardInput
    {

        public Keys RecentKey;

        // private TextBox _cmd_line_control;
        public bool CmdHasFocus
        {
            get { return false; } // _cmd_line_control.Focused; }
        }

        public KeyboardInput()
        {
            // _cmd_line_control = control;
        }

        public KeyboardAction ProcessKeydown(Keys key)
        {
            RecentKey = key;
            if (CmdHasFocus)
            {
                if (key == Keys.Enter)
                {
                    return KeyboardAction.RunCommand;
                }
                // otherwise do not do anything while the user is typing
                return KeyboardAction.None;
            }
            // else the user is talking to the 3D viewport
            switch (key)
            {
                case Keys.I:
                case Keys.O:
                case Keys.K:
                case Keys.L:
                case Keys.Oemcomma:
                case Keys.OemPeriod:
                    return KeyboardAction.DoRotation;
                case Keys.A:
                case Keys.D:
                case Keys.S:
                case Keys.W:
                case Keys.E:
                case Keys.Q:
                    return KeyboardAction.DoMotion;
                default:
                    return KeyboardAction.None;
            }
        }

        public Vector3 GetRotationAxis(Keys key)
        {
            switch (key)
            {
                // x axis
                case Keys.I: return Vector3.UnitX;
                case Keys.O: return -1 * Vector3.UnitX;
                // y axis
                case Keys.K: return Vector3.UnitY;
                case Keys.L: return -1 * Vector3.UnitY;
                // z axis
                case Keys.Oemcomma: return Vector3.UnitZ;
                case Keys.OemPeriod: return -1 * Vector3.UnitZ;
                //
                default:
                    Debug.Assert(false);
                    break;
            }
            return new Vector3(float.NaN, float.NaN, float.NaN);
        }

        public Vector3 GetDirectionNormalized(Keys key)
        {
            switch (key)
            {
                case Keys.A:
                    return new Vector3(-1, 0, 0);
                case Keys.D:
                    return new Vector3(1, 0, 0);
                case Keys.W:
                    return new Vector3(0, 1, 0);
                case Keys.S:
                    return new Vector3(0, -1, 0);
                case Keys.E:
                    return new Vector3(0, 0, -1);
                case Keys.Q:
                    return new Vector3(0, 0, 1);
                default:
                    Debug.Assert(false);
                    break;
            }
            return new Vector3(float.NaN, float.NaN, float.NaN);
        }

    }
}

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

namespace WinFormAnimation2D
```

```
{
    public enum Level
    {
        Debug = 0,
        Info = 1,
        Warn = 2,
        Error = 3,
        MAX
    }

    class Logger
    {
        private static string _local_app_data_dir = System.Environment.GetFolderPath(
                System.Environment.SpecialFolder.LocalApplicationData);

        private string[] _map_lvl2prefix = new string[]{ "Debug:  "
                                                      , "Info:   "
                                                      , "Warning:"
                                                      , "Error:  "
        };

        private string _log_file_path;
        public Logger(string file_name)
        {
            _log_file_path = System.IO.Path.Combine(_local_app_data_dir, file_name);
        }

        public void ClearLog()
        {
            System.IO.File.Delete(_log_file_path);
        }

        private void AppendLog(string text)
        {
            System.IO.File.AppendAllText(_log_file_path, string.Format("{0}\r\n", text));
        }

        public void Log(Level level, string message)
        {
            string head = _map_lvl2prefix[(int)level];
            AppendLog(head + message);
        }

        public void Log(string message)
        {
            // add default verbosity level
            Log(Level.Info, message);
        }

    }
}

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;
    }
}

/// <summary>
/// Intercept arrow keys to send input to the picture box.
/// (for the active control to see the keypress, return false)
/// </summary>
protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
{
    KeyboardAction action = _kbd.ProcessKeydown(keyData);
    if (action == KeyboardAction.DoRotation)
    {
        Vector3 rotation_axis = _kbd.GetRotationAxis(_kbd.RecentKey);
        _camera.RotateAround(rotation_axis);
        return true;
    }
    else if (action == KeyboardAction.DoMotion)
    {
        Vector3 direction = _kbd.GetDirectionNormalized(_kbd.RecentKey);
        _camera.MoveBy(direction);
        this.toolStripStatusLabel_camera_position.Text = _camera.GetTranslation.ToString();
        return true; // hide this key event from other controls
    }
    else if (action == KeyboardAction.RunCommand)
    {
        // _cmd.RunCmd(this.textBox_cli.Text);
        return true;
    }
    else if (action == KeyboardAction.None)
    {
        return base.ProcessCmdKey(ref msg, keyData);
    }
    Debug.Assert(false, "You forgot to handle some keyboard action");
    return false;
}

/// <summary>
/// Initialise the side tree view to show the scene.
/// </summary>
private void InitFillTreeFromWorldSingleEntity()
{
    this.treeView_entity_info.Nodes.Clear();
    // make root node and build whole tree
    var root_nd = new SceneTreeNode("root");
    // make entity tree
    var ent_one = new EntityTreeNode(_world._enttity_one.Name);
    ent_one.DrawMeshBounds = new BoundingBoxGroup(_world._enttity_one._extra_geometry._mesh_id2box.Values);
    // make entity mesh
    MeshTreeNode ent_mesh_nodes = MakeMeshTree(_world._enttity_one, _world._enttity_one._node);
    // make entity armature
    ArmatureTreeNode ent_arma_nodes = MakeArmatureTree(_world._enttity_one, _world._enttity_one._armature);
    root_nd.Nodes.Add(ent_one);
    ent_one.Nodes.Add(ent_mesh_nodes);
    ent_one.Nodes.Add(ent_arma_nodes);
    ent_arma_nodes.BackColor = Color.LightBlue;
    ent_mesh_nodes.BackColor = Color.LightGreen;
    ent_one.BackColor = Color.Gold;
    // attach and refresh
    this.treeView_entity_info.Nodes.Add(root_nd);
    // show the entity node
    // ent_one.EnsureVisible();
    this.treeView_entity_info.ExpandAll();
    ent_arma_nodes.EnsureVisible();
    this.treeView_entity_info.Invalidate();
}

private MeshTreeNode MakeMeshTree(Entity ent, Node nd)
{
    var current = new MeshTreeNode(nd.Name);
    var child_boxes = new List<MeshBounds>();
    if (nd.MeshCount > 1)
    {
        foreach (int mesh_id in nd.MeshIndices)
        {
            MeshBounds aabb = ent._extra_geometry._mesh_id2box[mesh_id];
            string mesh_name = _world._cur_scene._inner.Meshes[mesh_id].Name;
            var mesh_view_nd = new MeshTreeNode(mesh_name);
            var list = new List<MeshBounds>() { aabb };
            mesh_view_nd.DrawData = new BoundingBoxGroup(list);
            child_boxes.Add(aabb);
            current.Nodes.Add(mesh_view_nd);
        }
        // get a bounding box that covers all of the meshes assigned to this node
        current.DrawData = new BoundingBoxGroup(child_boxes);
    }
    else
    {
        // Place the bounding box of mesh as self bounding box
        MeshBounds aabb = ent._extra_geometry._mesh_id2box[nd.MeshIndices[0]];
        var list = new List<MeshBounds>() { aabb };
        current.DrawData = new BoundingBoxGroup(list);
    }
    foreach (var child_nd in nd.Children)
    {
        var treeview_child = MakeMeshTree(ent, child_nd);
        current.Nodes.Add(treeview_child);
    }
    return current;
}

private ArmatureTreeNode MakeArmatureTree(Entity ent, BoneNode nd)
{
    var current = new ArmatureTreeNode(nd._inner.Name);
    current.DrawData = ent._extra_geometry._bone_id2triangle[nd._inner.Name];
    foreach (var child_nd in nd.Children)
    {
        var treeview_child = MakeArmatureTree(ent, child_nd);
        current.Nodes.Add(treeview_child);
    }
    return current;
}

private void HighlightSlectedNode()
{
    var view_nd = (IHighlightableNode)this.treeView_entity_info.SelectedNode;
    if (view_nd != null)
    {
        last_selected_node = view_nd;
        view_nd.Render();
    }
    // last_selected_node is null only on scene load
    else if (last_selected_node != null)
    {
        last_selected_node.Render();
    }
}

private void PrepareOpenGLRenderFrame()
{
    // guard if GLControl has not loaded yet
    if (! LoadOpenGLDone)
    {
        return;
    }
    _world._renderer.ClearOpenglFrameForRender(_camera.MatrixToOpenGL());
    if (Properties.Settings.Default.OpenGLDrawAxis)
    {
        _world._renderer.DrawAxis3D();
    }

    UpdateFrame();

    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    GL.Color3(Color.Green);
}

private void RenderBones(Entity ent)
{
    foreach (var bounds in ent._extra_geometry._bone_id2triangle.Values)
    {
        bounds.Render(Pens.Black);
    }
}

// use unix style command invocation
// cmdname arg1 arg2 arg3
private void button_RunCli_Click(object sender, EventArgs e)
{
    // this._cmd.RunCmd(this.textBox_cli.Text);
}

private void trackBar_AnimationTime_ValueChanged(object sender, EventArgs e)
{
    if (Current == null)
    {
        return;
    }
    // if the user changed the value
    if (this.trackBar_time.Focused)
    {
        double factor = Current._action.TotalDurationSeconds / TrackBarTimeRange;
        double time_seconds = (sender as TrackBar).Value * factor;
        Current._action.SetTime(time_seconds);
        _world._action_one.ApplyAnimation(Current._armature
            , Current._action);
        this.toolStripStatusLabel_AnimTime.Text = time_seconds.ToString("F4");
    }
}
```

```
private void checkBox_renderBones_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.RenderAllBoneBounds = this.checkBox_renderBones.Checked;
    // this._cmd.RunCmd("set RenderAllBoneBounds " + this.checkBox_renderBones.Checked);
}

private void checkBox_render_boxes_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.RenderAllMeshBounds = this.checkBox_render_boxes.Checked;
    // this._cmd.RunCmd("set RenderAllMeshBounds " + this.checkBox_render_boxes.Checked);
}

private void checkBox_breakpoints_on_CheckedChanged(object sender, EventArgs e)
{
    Breakpoints.Allow = this.checkBox_breakpoints_on.Checked;
}

private void checkBox_triangulate_CheckedChanged(object sender, EventArgs e)
{
    //Properties.Settings.Default.TriangulateMesh = this.checkBox_triangulate.Checked;
    // this._cmd.RunCmd("set TriangulateMesh " + this.checkBox_triangulate.Checked);
}

private void checkBox_moveCamera_CheckedChanged(object sender, EventArgs e)
{
    //Properties.Settings.Default.MoveCamera = this.checkBox_moveCamera.Checked;
    // this._cmd.RunCmd("set MoveCamera " + this.checkBox_triangulate.Checked);
}

private void checkBox_RenderNormals_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.RenderNormals = this.checkBox_RenderNormals.Checked;
}

private void checkBox_OrbitingCamera_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.OrbitingCamera = this.checkBox_OrbitingCamera.Checked;
    button_ResetCamera_Click(null, null);
}

private void checkBox_FixCameraPlane_CheckedChanged(object sender, EventArgs e)
{
    //Properties.Settings.Default.FixCameraPlane = this.checkBox_FixCameraPlane.Checked;
}

private void MainForm_ResizeEnd(object sender, EventArgs e)
{
    _world._renderer.ResizeOpenGL(this.glControl1.Width, this.glControl1.Height);
}

private void glControl1_Load(object sender, EventArgs e)
{
    _world._renderer.InitOpenGL();
    _world._renderer.ResizeOpenGL(this.glControl1.Width, this.glControl1.Height);
    LoadOpenGLDone = true;
    // register Idle event so we get regular callbacks for drawing
    Application.Idle += ApplicationIdle;
}

private void ApplicationIdle(object sender, EventArgs e)
{
    if(this.IsDisposed)
    {
        return;
    }
    while (glControl1.IsIdle)
    {
        UpdateFrame();
        RenderFrame();
    }
}

private void RenderFrame()
{
    PrepareOpenGLRenderFrame();
    // render entity
    if (! _world.HasScene)
    {
        glControl1.SwapBuffers();
        return;
    }
    _world.RenderWorld();
    // currently selected in tree view
    // Disable depth test because we want bones to always be visible
    GL.Disable(EnableCap.DepthTest);
    if (Current != null)
    {
        Current._extra_geometry.UpdateBonePositions(Current._armature);
        if (Properties.Settings.Default.RenderAllBoneBounds)
        {
            RenderBones(Current);
        }
    }
    HighlightSlectedNode();
    glControl1.SwapBuffers();
    // picture box was not made for such fast updates, we will update it with a timer
    // enable to see the slow speed of OpenGL update
    // glControl1.SwapBuffers();
}

private void UpdateFrame()
{
    this.toolStripStatusLabel_mouse_coords.Text = _mouse.InnerWorldPos.ToString();
    LastFrameDelay = _last_frame_sw.ElapsedMilliseconds;
    _last_frame_sw.Restart();
    _world.Update(LastFrameDelay);
}

private void glControl1_MouseDown(object sender, MouseEventArgs e)
{
    _mouse.RecordMouseClick(e);
    _mouse.RecordInnerWorldMouseClick(_camera.ConvertScreen2WorldCoordinates(_mouse.ClickPos));
    // this.toolStripStatusLabel_entity_position.Text = Current.GetTranslation.ToString();
    // this.treeView_entity_info.SelectedNode = this.treeView_entity_info.Nodes[Current.Name];
}

private void glControl1_MouseMove(object sender, MouseEventArgs e)
{
    _mouse.RecordMouseMove(e);
    _mouse.RecordInnerWorldMouseMove(_camera.ConvertScreen2WorldCoordinates(_mouse.CurrentPos));
    if (_world.CheckMouseEntitySelect(_mouse))
    {
        //this.toolStripStatusLabel_is_selected.Text = "HAS ENTITY";
    }
    else
    {
        //this.toolStripStatusLabel_is_selected.Text = "___empty___";
    }

    // Process mouse motion only if it is pressed
    if (! _mouse.IsPressed) {
        return;
    }
    // time to do some rotation
    _camera.OnMouseMove(_mouse.FrameDelta.X, _mouse.FrameDelta.Y);
    this.toolStripStatusLabel_is_selected.Text = _mouse.FrameDelta.ToString();
}

private void glControl1_MouseUp(object sender, MouseEventArgs e)
{
    _mouse.IsPressed = false;
}

private void glControl1_MouseWheel(object sender, MouseEventArgs e)
{
    _camera.Scroll(Math.Sign(e.Delta));
}

private void button_ResetCamera_Click(object sender, EventArgs e)
{
    Matrix4 opengl_camera_init = Matrix4.LookAt(0, 50, 500, 0, 0, 0, 0, 1, 0).Inverted();
    _camera = new CameraDevice(opengl_camera_init);
}

private void checkBox_playall_CheckedChanged(object sender, EventArgs e)
{
    _cmd.playall(this.checkBox_playall.Checked);
}

private void checkBox_OpenGL_Material_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.OpenGLMaterial = this.checkBox_OpenGL_Material.Checked;
}
```

```
private void button_step_frame_Click(object sender, EventArgs e)
{
    _cmd.stepall();
}

private void button_back_one_frame_Click(object sender, EventArgs e)
{
    _cmd.bkf();
}

private void checkBox_OpenGLDrawAxis_CheckedChanged(object sender, EventArgs e)
{
    Properties.Settings.Default.OpenGLDrawAxis = this.checkBox_OpenGLDrawAxis.Checked;
}

private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
  MessageBox.Show("Курсовая работа \n \"Программа скелетная анимация\" \n Выполнил студент БПИ 151 \n Абрамов Артем");
}

private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
{
    Properties.Settings.Default.RenderNormals = false;
    Properties.Settings.Default.RenderAllBoneBounds = false;
    Properties.Settings.Default.OpenGLMaterial = false;
    Properties.Settings.Default.OpenGLDrawAxis = true;
    Properties.Settings.Default.Save();
}

/// <summary>
/// Open file, read and verify data
/// </summary>
private void OpenFileCollada(string filepath)
{
    try {
        byte[] data = File.ReadAllBytes(filepath);
        _world.LoadScene(data);
        // we have to wait for OpenGL to load before uploading VBOs to OpenGL server
        _world._enttity_one.UploadMeshVBO(_world._cur_scene._inner.Materials);
        _cmd._current = _world._enttity_one;
        InitFillTreeFromWorldSingleEntity();
        Recent.CurrentlyOpenFilePath = filepath;
        // add to open recent
        Recent.AddRecentFile(filepath);
        RefreshOpenRecentMenu();
        this.treeView_entity_info.SelectedNode = null;
        last_selected_node = null;
        this.toolStripStatusLabel_AnimTime.Text = "";
    }
    catch (Exception ex) {
        MessageBox.Show("Sorry, the file format is invalid.");
        return;
    }
}

/// <summary>
/// Show open file dialog to choose csv file.
/// </summary>
private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    string filepath = OpenFileDialogGetPath();
    if (filepath == null) {
        return;
    }
    Properties.Settings.Default.RecentDirectory = Path.GetDirectoryName(filepath);
    OpenFileCollada(filepath);
}

/// <summary>
/// Opens a dialog to get path of file to open from te user.
/// </summary>
public string OpenFileDialogGetPath()
{
    OpenFileDialog file_dialog = new OpenFileDialog
    {
        InitialDirectory = Properties.Settings.Default.RecentDirectory,
        Filter = "Collada files (*.dae)|*.dae|All files (*.*)|*.*",
        FilterIndex = 0,
        RestoreDirectory = true,
        Title = "Select a collada file...",
    };
```

```
            if (file_dialog.ShowDialog() == DialogResult.OK) {
                return file_dialog.FileName;
            }
            return null;
        }

    }
}

namespace WinFormAnimation2D
{
    partial class MainForm
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        System.ComponentModel.ComponentResourceManager resources = new System.ComponentModel.ComponentResourceManager(typeof(MainForm
            this.statusStrip1 = new System.Windows.Forms.StatusStrip();
            this.toolStripStatusLabel1 = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel_is_selected = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel_mouse_coords = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel2 = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel_camera_position = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel3 = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel_entity_position = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel4 = new System.Windows.Forms.ToolStripStatusLabel();
            this.toolStripStatusLabel_AnimTime = new System.Windows.Forms.ToolStripStatusLabel();
            this.trackBar_time = new System.Windows.Forms.TrackBar();
            this.label3 = new System.Windows.Forms.Label();
            this.glControl1 = new OpenTK.GLControl();
            this.menuStrip1 = new System.Windows.Forms.MenuStrip();
            this.tabPage_RenderOptions = new System.Windows.Forms.TabPage();
            this.checkBox_render_boxes = new System.Windows.Forms.CheckBox();
            this.checkBox_renderBones = new System.Windows.Forms.CheckBox();
            this.button_ResetCamera = new System.Windows.Forms.Button();
            this.checkBox_breakpoints_on = new System.Windows.Forms.CheckBox();
            this.checkBox_OrbitingCamera = new System.Windows.Forms.CheckBox();
            this.checkBox_RenderNormals = new System.Windows.Forms.CheckBox();
            this.checkBox_playall = new System.Windows.Forms.CheckBox();
            this.checkBox_OpenGL_Material = new System.Windows.Forms.CheckBox();
            this.button_step_frame = new System.Windows.Forms.Button();
            this.checkBox_OpenGLDrawAxis = new System.Windows.Forms.CheckBox();
            this.tabPage_TreeView = new System.Windows.Forms.TabPage();
            this.treeView_entity_info = new System.Windows.Forms.TreeView();
            this.label2 = new System.Windows.Forms.Label();
            this.tabControl_panel = new System.Windows.Forms.TabControl();
            this.button_back_one_frame = new System.Windows.Forms.Button();
            this.fileToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.newToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.openToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.toolStripSeparator2 = new System.Windows.Forms.ToolStripSeparator();
            this.exitToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.helpToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.aboutToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.recentToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.statusStrip1.SuspendLayout();
            ((System.ComponentModel.ISupportInitialize)(this.trackBar_time)).BeginInit();
            this.menuStrip1.SuspendLayout();
            this.tabPage_RenderOptions.SuspendLayout();
this.tabPage_TreeView.SuspendLayout();
this.tabControl_panel.SuspendLayout();
this.SuspendLayout();
//
// statusStrip1
//
this.statusStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
this.toolStripStatusLabel1,
this.toolStripStatusLabel_is_selected,
this.toolStripStatusLabel_mouse_coords,
this.toolStripStatusLabel2,
this.toolStripStatusLabel_camera_position,
this.toolStripStatusLabel3,
this.toolStripStatusLabel_entity_position,
this.toolStripStatusLabel4,
this.toolStripStatusLabel_AnimTime});
this.statusStrip1.Location = new System.Drawing.Point(0, 506);
this.statusStrip1.Name = "statusStrip1";
this.statusStrip1.Size = new System.Drawing.Size(958, 22);
this.statusStrip1.TabIndex = 25;
this.statusStrip1.Text = "statusStrip1";
//
// toolStripStatusLabel1
//
this.toolStripStatusLabel1.AutoSize = false;
this.toolStripStatusLabel1.Name = "toolStripStatusLabel1";
this.toolStripStatusLabel1.Size = new System.Drawing.Size(46, 19);
this.toolStripStatusLabel1.Text = "mouse:";
this.toolStripStatusLabel1.Visible = false;
//
// toolStripStatusLabel_is_selected
//
this.toolStripStatusLabel_is_selected.AutoSize = false;
this.toolStripStatusLabel_is_selected.Name = "toolStripStatusLabel_is_selected";
this.toolStripStatusLabel_is_selected.Size = new System.Drawing.Size(122, 19);
this.toolStripStatusLabel_is_selected.Text = "toolStripStatusLabel1";
this.toolStripStatusLabel_is_selected.Visible = false;
//
// toolStripStatusLabel_mouse_coords
//
this.toolStripStatusLabel_mouse_coords.AutoSize = false;
this.toolStripStatusLabel_mouse_coords.BorderSides = System.Windows.Forms.ToolStripStatusLabelBorderSides.Right;
this.toolStripStatusLabel_mouse_coords.Name = "toolStripStatusLabel_mouse_coords";
this.toolStripStatusLabel_mouse_coords.Size = new System.Drawing.Size(140, 19);
this.toolStripStatusLabel_mouse_coords.Text = "toolStripStatusLabel2";
this.toolStripStatusLabel_mouse_coords.Visible = false;
//
// toolStripStatusLabel2
//
this.toolStripStatusLabel2.Name = "toolStripStatusLabel2";
this.toolStripStatusLabel2.Size = new System.Drawing.Size(49, 19);
this.toolStripStatusLabel2.Text = "camera:";
this.toolStripStatusLabel2.Visible = false;
//
// toolStripStatusLabel_camera_position
//
this.toolStripStatusLabel_camera_position.AutoSize = false;
this.toolStripStatusLabel_camera_position.BorderSides = System.Windows.Forms.ToolStripStatusLabelBorderSides.Right;
this.toolStripStatusLabel_camera_position.Name = "toolStripStatusLabel_camera_position";
this.toolStripStatusLabel_camera_position.Size = new System.Drawing.Size(118, 19);
this.toolStripStatusLabel_camera_position.Text = "toolStripStatusLabel1";
this.toolStripStatusLabel_camera_position.Visible = false;
//
// toolStripStatusLabel3
//
this.toolStripStatusLabel3.Name = "toolStripStatusLabel3";
this.toolStripStatusLabel3.Size = new System.Drawing.Size(40, 19);
this.toolStripStatusLabel3.Text = "entity:";
this.toolStripStatusLabel3.Visible = false;
//
// toolStripStatusLabel_entity_position
//
this.toolStripStatusLabel_entity_position.AutoSize = false;
this.toolStripStatusLabel_entity_position.BorderSides = System.Windows.Forms.ToolStripStatusLabelBorderSides.Right;
this.toolStripStatusLabel_entity_position.Name = "toolStripStatusLabel_entity_position";
this.toolStripStatusLabel_entity_position.Size = new System.Drawing.Size(118, 19);
this.toolStripStatusLabel_entity_position.Text = "toolStripStatusLabel1";
this.toolStripStatusLabel_entity_position.Visible = false;
//
// toolStripStatusLabel4
//
this.toolStripStatusLabel4.Name = "toolStripStatusLabel4";
```

```
    this.toolStripStatusLabel4.Size = new System.Drawing.Size(34, 17);
    this.toolStripStatusLabel4.Text = "time:";
    //
    // toolStripStatusLabel_AnimTime
    //
    this.toolStripStatusLabel_AnimTime.Name = "toolStripStatusLabel_AnimTime";
    this.toolStripStatusLabel_AnimTime.Size = new System.Drawing.Size(118, 17);
    this.toolStripStatusLabel_AnimTime.Text = "toolStripStatusLabel4";
    //
    // trackBar_time
    //
    this.trackBar_time.Location = new System.Drawing.Point(88, 38);
    this.trackBar_time.Maximum = 20;
    this.trackBar_time.Name = "trackBar_time";
    this.trackBar_time.Size = new System.Drawing.Size(578, 45);
    this.trackBar_time.TabIndex = 36;
    this.trackBar_time.ValueChanged += new System.EventHandler(this.trackBar_AnimationTime_ValueChanged);
    //
    // label3
    //
    this.label3.AutoSize = true;
    this.label3.Location = new System.Drawing.Point(34, 38);
    this.label3.Name = "label3";
    this.label3.Size = new System.Drawing.Size(48, 13);
    this.label3.TabIndex = 37;
    this.label3.Text = "Time bar";
    //
    // glControl1
    //
  this.glControl1.Anchor = ((System.Windows.Forms.AnchorStyles)(((System.Windows.Forms.AnchorStyles.Top | System.Windows.Forms.AnchorSt
    | System.Windows.Forms.AnchorStyles.Left)));
    this.glControl1.BackColor = System.Drawing.Color.Black;
    this.glControl1.Location = new System.Drawing.Point(12, 78);
    this.glControl1.Name = "glControl1";
    this.glControl1.Size = new System.Drawing.Size(721, 423);
    this.glControl1.TabIndex = 47;
    this.glControl1.VSync = true;
    this.glControl1.Load += new System.EventHandler(this.glControl1_Load);
    this.glControl1.MouseDown += new System.Windows.Forms.MouseEventHandler(this.glControl1_MouseDown);
    this.glControl1.MouseMove += new System.Windows.Forms.MouseEventHandler(this.glControl1_MouseMove);
    this.glControl1.MouseUp += new System.Windows.Forms.MouseEventHandler(this.glControl1_MouseUp);
    //
    // menuStrip1
    //
    this.menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
    this.fileToolStripMenuItem,
    this.helpToolStripMenuItem});
    this.menuStrip1.Location = new System.Drawing.Point(0, 0);
    this.menuStrip1.Name = "menuStrip1";
    this.menuStrip1.Size = new System.Drawing.Size(958, 24);
    this.menuStrip1.TabIndex = 51;
    this.menuStrip1.Text = "menuStrip1";
    //
    // tabPage_RenderOptions
    //
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_OpenGLDrawAxis);
    this.tabPage_RenderOptions.Controls.Add(this.button_back_one_frame);
    this.tabPage_RenderOptions.Controls.Add(this.button_step_frame);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_OpenGL_Material);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_playall);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_RenderNormals);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_OrbitingCamera);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_breakpoints_on);
    this.tabPage_RenderOptions.Controls.Add(this.button_ResetCamera);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_renderBones);
    this.tabPage_RenderOptions.Controls.Add(this.checkBox_render_boxes);
    this.tabPage_RenderOptions.Location = new System.Drawing.Point(4, 22);
    this.tabPage_RenderOptions.Name = "tabPage_RenderOptions";
    this.tabPage_RenderOptions.Padding = new System.Windows.Forms.Padding(3);
    this.tabPage_RenderOptions.Size = new System.Drawing.Size(211, 397);
    this.tabPage_RenderOptions.TabIndex = 1;
    this.tabPage_RenderOptions.Text = "Render";
    this.tabPage_RenderOptions.UseVisualStyleBackColor = true;
    //
    // checkBox_render_boxes
    //
    this.checkBox_render_boxes.AutoSize = true;
    this.checkBox_render_boxes.Location = new System.Drawing.Point(19, 117);
    this.checkBox_render_boxes.Name = "checkBox_render_boxes";
    this.checkBox_render_boxes.Size = new System.Drawing.Size(93, 17);
    this.checkBox_render_boxes.TabIndex = 44;
    this.checkBox_render_boxes.Text = "Render Boxes";
    this.checkBox_render_boxes.UseVisualStyleBackColor = true;
    this.checkBox_render_boxes.Visible = false;
    this.checkBox_render_boxes.CheckedChanged += new System.EventHandler(this.checkBox_render_boxes_CheckedChanged);
    //
    // checkBox_renderBones
    //
    this.checkBox_renderBones.AutoSize = true;
    this.checkBox_renderBones.Location = new System.Drawing.Point(19, 140);
    this.checkBox_renderBones.Name = "checkBox_renderBones";
    this.checkBox_renderBones.Size = new System.Drawing.Size(94, 17);
    this.checkBox_renderBones.TabIndex = 43;
    this.checkBox_renderBones.Text = "Render Bones";
    this.checkBox_renderBones.UseVisualStyleBackColor = true;
    this.checkBox_renderBones.CheckedChanged += new System.EventHandler(this.checkBox_renderBones_CheckedChanged);
    //
    // button_ResetCamera
    //
    this.button_ResetCamera.Location = new System.Drawing.Point(19, 11);
    this.button_ResetCamera.Name = "button_ResetCamera";
    this.button_ResetCamera.Size = new System.Drawing.Size(75, 35);
    this.button_ResetCamera.TabIndex = 38;
    this.button_ResetCamera.Text = "Camera reset";
    this.button_ResetCamera.UseVisualStyleBackColor = true;
    this.button_ResetCamera.Click += new System.EventHandler(this.button_ResetCamera_Click);
    //
    // checkBox_breakpoints_on
    //
    this.checkBox_breakpoints_on.AutoSize = true;
    this.checkBox_breakpoints_on.Location = new System.Drawing.Point(19, 52);
    this.checkBox_breakpoints_on.Name = "checkBox_breakpoints_on";
    this.checkBox_breakpoints_on.Size = new System.Drawing.Size(118, 17);
    this.checkBox_breakpoints_on.TabIndex = 27;
    this.checkBox_breakpoints_on.Text = "Breakpoints On/Off";
    this.checkBox_breakpoints_on.UseVisualStyleBackColor = true;
    this.checkBox_breakpoints_on.Visible = false;
    this.checkBox_breakpoints_on.CheckedChanged += new System.EventHandler(this.checkBox_breakpoints_on_CheckedChanged);
    //
    // checkBox_OrbitingCamera
    //
    this.checkBox_OrbitingCamera.AutoSize = true;
    this.checkBox_OrbitingCamera.Location = new System.Drawing.Point(19, 163);
    this.checkBox_OrbitingCamera.Name = "checkBox_OrbitingCamera";
    this.checkBox_OrbitingCamera.Size = new System.Drawing.Size(101, 17);
    this.checkBox_OrbitingCamera.TabIndex = 50;
    this.checkBox_OrbitingCamera.Text = "Orbiting Camera";
    this.checkBox_OrbitingCamera.UseVisualStyleBackColor = true;
    this.checkBox_OrbitingCamera.CheckedChanged += new System.EventHandler(this.checkBox_OrbitingCamera_CheckedChanged);
    //
    // checkBox_RenderNormals
    //
    this.checkBox_RenderNormals.AutoSize = true;
    this.checkBox_RenderNormals.Location = new System.Drawing.Point(19, 76);
    this.checkBox_RenderNormals.Name = "checkBox_RenderNormals";
    this.checkBox_RenderNormals.Size = new System.Drawing.Size(122, 17);
    this.checkBox_RenderNormals.TabIndex = 51;
    this.checkBox_RenderNormals.Text = "Render with normals";
    this.checkBox_RenderNormals.UseVisualStyleBackColor = true;
    this.checkBox_RenderNormals.CheckedChanged += new System.EventHandler(this.checkBox_RenderNormals_CheckedChanged);
    //
    // checkBox_playall
    //
    this.checkBox_playall.AutoSize = true;
    this.checkBox_playall.Location = new System.Drawing.Point(19, 199);
    this.checkBox_playall.Name = "checkBox_playall";
    this.checkBox_playall.Size = new System.Drawing.Size(94, 17);
    this.checkBox_playall.TabIndex = 52;
    this.checkBox_playall.Text = "Play animation";
    this.checkBox_playall.UseVisualStyleBackColor = true;
    this.checkBox_playall.CheckedChanged += new System.EventHandler(this.checkBox_playall_CheckedChanged);
    //
    // checkBox_OpenGL_Material
    //
    this.checkBox_OpenGL_Material.AutoSize = true;
    this.checkBox_OpenGL_Material.Location = new System.Drawing.Point(19, 222);
    this.checkBox_OpenGL_Material.Name = "checkBox_OpenGL_Material";
    this.checkBox_OpenGL_Material.Size = new System.Drawing.Size(91, 17);
    this.checkBox_OpenGL_Material.TabIndex = 53;
    this.checkBox_OpenGL_Material.Text = "Apply material";
    this.checkBox_OpenGL_Material.UseVisualStyleBackColor = true;
this.checkBox_OpenGL_Material.CheckedChanged += new System.EventHandler(this.checkBox_OpenGL_Material_CheckedChanged);
    //
    // button_step_frame
    //
    this.button_step_frame.Location = new System.Drawing.Point(19, 245);
    this.button_step_frame.Name = "button_step_frame";
    this.button_step_frame.Size = new System.Drawing.Size(129, 23);
    this.button_step_frame.TabIndex = 54;
    this.button_step_frame.Text = "Small jump forward";
    this.button_step_frame.UseVisualStyleBackColor = true;
    this.button_step_frame.Click += new System.EventHandler(this.button_step_frame_Click);
    //
    // checkBox_OpenGLDrawAxis
    //
    this.checkBox_OpenGLDrawAxis.AutoSize = true;
    this.checkBox_OpenGLDrawAxis.Location = new System.Drawing.Point(19, 304);
    this.checkBox_OpenGLDrawAxis.Name = "checkBox_OpenGLDrawAxis";
    this.checkBox_OpenGLDrawAxis.Size = new System.Drawing.Size(89, 17);
    this.checkBox_OpenGLDrawAxis.TabIndex = 57;
    this.checkBox_OpenGLDrawAxis.Text = "Draw 3D axis";
    this.checkBox_OpenGLDrawAxis.UseVisualStyleBackColor = true;
this.checkBox_OpenGLDrawAxis.CheckedChanged += new System.EventHandler(this.checkBox_OpenGLDrawAxis_CheckedChanged);
    //
    // tabPage_TreeView
    //
    this.tabPage_TreeView.Controls.Add(this.label2);
    this.tabPage_TreeView.Controls.Add(this.treeView_entity_info);
    this.tabPage_TreeView.Location = new System.Drawing.Point(4, 22);
    this.tabPage_TreeView.Name = "tabPage_TreeView";
    this.tabPage_TreeView.Padding = new System.Windows.Forms.Padding(3);
    this.tabPage_TreeView.Size = new System.Drawing.Size(211, 397);
    this.tabPage_TreeView.TabIndex = 0;
    this.tabPage_TreeView.Text = "Tree";
    this.tabPage_TreeView.UseVisualStyleBackColor = true;
    //
    // treeView_entity_info
    //
this.treeView_entity_info.Anchor = ((System.Windows.Forms.AnchorStyles)((((System.Windows.Forms.AnchorStyles.Top | System.Windows.For
     | System.Windows.Forms.AnchorStyles.Left)
     | System.Windows.Forms.AnchorStyles.Right)));
    this.treeView_entity_info.Location = new System.Drawing.Point(6, 19);
    this.treeView_entity_info.Name = "treeView_entity_info";
    this.treeView_entity_info.Size = new System.Drawing.Size(199, 372);
    this.treeView_entity_info.TabIndex = 26;
    //
    // label2
    //
    this.label2.AutoSize = true;
    this.label2.Location = new System.Drawing.Point(6, 3);
    this.label2.Name = "label2";
    this.label2.Size = new System.Drawing.Size(122, 13);
    this.label2.TabIndex = 22;
    this.label2.Text = "Currently selected entity:";
    //
    // tabControl_panel
    //
this.tabControl_panel.Anchor = ((System.Windows.Forms.AnchorStyles)((((System.Windows.Forms.AnchorStyles.Top | System.Windows.Forms.A
     | System.Windows.Forms.AnchorStyles.Left)
     | System.Windows.Forms.AnchorStyles.Right)));
    this.tabControl_panel.Controls.Add(this.tabPage_TreeView);
    this.tabControl_panel.Controls.Add(this.tabPage_RenderOptions);
    this.tabControl_panel.Location = new System.Drawing.Point(739, 78);
    this.tabControl_panel.Name = "tabControl_panel";
    this.tabControl_panel.SelectedIndex = 0;
    this.tabControl_panel.Size = new System.Drawing.Size(219, 423);
    this.tabControl_panel.TabIndex = 50;
    //
    // button_back_one_frame
    //
    this.button_back_one_frame.Location = new System.Drawing.Point(19, 274);
    this.button_back_one_frame.Name = "button_back_one_frame";
    this.button_back_one_frame.Size = new System.Drawing.Size(129, 23);
    this.button_back_one_frame.TabIndex = 56;
    this.button_back_one_frame.Text = "Play back one keyframe";
    this.button_back_one_frame.UseVisualStyleBackColor = true;
    this.button_back_one_frame.Visible = false;
    this.button_back_one_frame.Click += new System.EventHandler(this.button_back_one_frame_Click);
    //
    // fileToolStripMenuItem
    //
    this.fileToolStripMenuItem.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[] {
    this.newToolStripMenuItem,
    this.openToolStripMenuItem,
    this.recentToolStripMenuItem,
    this.toolStripSeparator2,
  this.exitToolStripMenuItem});
  this.fileToolStripMenuItem.Name = "fileToolStripMenuItem";
  this.fileToolStripMenuItem.Size = new System.Drawing.Size(37, 20);
  this.fileToolStripMenuItem.Text = "&File";
  //
  // newToolStripMenuItem
  //
  this.newToolStripMenuItem.Image = ((System.Drawing.Image)(resources.GetObject("newToolStripMenuItem.Image")));
  this.newToolStripMenuItem.ImageTransparentColor = System.Drawing.Color.Magenta;
  this.newToolStripMenuItem.Name = "newToolStripMenuItem";
this.newToolStripMenuItem.ShortcutKeys = ((System.Windows.Forms.Keys)((System.Windows.Forms.Keys.Control | System.Windows.Forms.Key
  this.newToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
  this.newToolStripMenuItem.Text = "&New";
  //
  // openToolStripMenuItem
  //
  this.openToolStripMenuItem.Image = ((System.Drawing.Image)(resources.GetObject("openToolStripMenuItem.Image")));
  this.openToolStripMenuItem.ImageTransparentColor = System.Drawing.Color.Magenta;
  this.openToolStripMenuItem.Name = "openToolStripMenuItem";
this.openToolStripMenuItem.ShortcutKeys = ((System.Windows.Forms.Keys)((System.Windows.Forms.Keys.Control | System.Windows.Forms.Ke
  this.openToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
  this.openToolStripMenuItem.Text = "&Open";
  this.openToolStripMenuItem.Click += new System.EventHandler(this.openToolStripMenuItem_Click);
  //
  // toolStripSeparator2
  //
  this.toolStripSeparator2.Name = "toolStripSeparator2";
  this.toolStripSeparator2.Size = new System.Drawing.Size(149, 6);
  //
  // exitToolStripMenuItem
  //
  this.exitToolStripMenuItem.Name = "exitToolStripMenuItem";
  this.exitToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
  this.exitToolStripMenuItem.Text = "E&xit";
  //
  // helpToolStripMenuItem
  //
  this.helpToolStripMenuItem.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[] {
  this.aboutToolStripMenuItem});
  this.helpToolStripMenuItem.Name = "helpToolStripMenuItem";
  this.helpToolStripMenuItem.Size = new System.Drawing.Size(44, 20);
  this.helpToolStripMenuItem.Text = "&Help";
  //
  // aboutToolStripMenuItem
  //
  this.aboutToolStripMenuItem.Name = "aboutToolStripMenuItem";
  this.aboutToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
  this.aboutToolStripMenuItem.Text = "&About...";
  this.aboutToolStripMenuItem.Click += new System.EventHandler(this.aboutToolStripMenuItem_Click);
  //
  // recentToolStripMenuItem
  //
  this.recentToolStripMenuItem.Image = ((System.Drawing.Image)(resources.GetObject("recentToolStripMenuItem.Image")));
  this.recentToolStripMenuItem.Name = "recentToolStripMenuItem";
  this.recentToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
  this.recentToolStripMenuItem.Text = "Open &Recent";
  //
  // MainForm
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(958, 528);
  this.Controls.Add(this.tabControl_panel);
  this.Controls.Add(this.glControl1);
  this.Controls.Add(this.label3);
  this.Controls.Add(this.statusStrip1);
  this.Controls.Add(this.menuStrip1);
  this.Controls.Add(this.trackBar_time);
  this.MainMenuStrip = this.menuStrip1;
  this.Name = "MainForm";
  this.Text = "Form1";
  this.FormClosing += new System.Windows.Forms.FormClosingEventHandler(this.MainForm_FormClosing);
  this.ResizeEnd += new System.EventHandler(this.MainForm_ResizeEnd);
  this.statusStrip1.ResumeLayout(false);
  this.statusStrip1.PerformLayout();
  ((System.ComponentModel.ISupportInitialize)(this.trackBar_time)).EndInit();
  this.menuStrip1.ResumeLayout(false);
  this.menuStrip1.PerformLayout();
  this.tabPage_RenderOptions.ResumeLayout(false);
  this.tabPage_RenderOptions.PerformLayout();
  this.tabPage_TreeView.ResumeLayout(false);
  this.tabPage_TreeView.PerformLayout();
```

```
            this.tabControl_panel.ResumeLayout(false);
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion
        private System.Windows.Forms.StatusStrip statusStrip1;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel_is_selected;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel_mouse_coords;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel_camera_position;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel1;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel2;
        private System.Windows.Forms.TrackBar trackBar_time;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel3;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel_entity_position;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel4;
        private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabel_AnimTime;
        private OpenTK.GLControl glControl1;
        private System.Windows.Forms.MenuStrip menuStrip1;
        private System.Windows.Forms.TabPage tabPage_RenderOptions;
        private System.Windows.Forms.CheckBox checkBox_OpenGLDrawAxis;
        private System.Windows.Forms.Button button_back_one_frame;
        private System.Windows.Forms.Button button_step_frame;
        private System.Windows.Forms.CheckBox checkBox_OpenGL_Material;
        private System.Windows.Forms.CheckBox checkBox_playall;
        private System.Windows.Forms.CheckBox checkBox_RenderNormals;
        private System.Windows.Forms.CheckBox checkBox_OrbitingCamera;
        private System.Windows.Forms.CheckBox checkBox_breakpoints_on;
        private System.Windows.Forms.Button button_ResetCamera;
        private System.Windows.Forms.CheckBox checkBox_renderBones;
        private System.Windows.Forms.CheckBox checkBox_render_boxes;
        private System.Windows.Forms.TabPage tabPage_TreeView;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.TreeView treeView_entity_info;
        private System.Windows.Forms.TabControl tabControl_panel;
        private System.Windows.Forms.ToolStripMenuItem fileToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem newToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem openToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem recentToolStripMenuItem;
        private System.Windows.Forms.ToolStripSeparator toolStripSeparator2;
        private System.Windows.Forms.ToolStripMenuItem exitToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem helpToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem aboutToolStripMenuItem;
    }
}


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;
using OpenTK;
using OpenTK.Graphics.OpenGL;
using OpenTK.Graphics;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace WinFormAnimation2D
{

    struct Vbo
    {
        public int VertexBufferId;
        public int ColorBufferId;
        public int TexCoordBufferId;
        public int NormalBufferId;
        public int ElementBufferId;
        public int NumIndices;
    }

    /// <summary>
    /// Mesh rendering using VBOs.
    /// Based on http://www.opentk.com/files/T08_VBO.cs
    /// </summary>
    class MeshDraw
```

```
{
    public Mesh _mesh;
    public Dictionary<int,Matrix4x4> _vertex_id2matrix = new Dictionary<int,Matrix4x4>();
    public Vbo _vbo;
    public Material _material;
    public int _apply_material_id;

    /// <summary>
    /// Uploads the data to the GPU.
    /// </summary>
    public MeshDraw(Mesh mesh, IList<Material> materials)
    {
        Debug.Assert(mesh != null);
        _mesh = mesh;
        _material = materials[mesh.MaterialIndex];
        Upload(out _vbo);
        _apply_material_id = CompileMaterialDisplayList();
    }

    /// <summary>
    /// Render mesh from GPU memory.
    /// </summary>
    public void RenderVBO()
    {
        GL.PushClientAttrib(ClientAttribMask.ClientVertexArrayBit);
        Debug.Assert(_vbo.VertexBufferId != 0);
        Debug.Assert(_vbo.ElementBufferId != 0);
        // material
        if (Properties.Settings.Default.OpenGLMaterial)
        {
            GL.CallList(_apply_material_id);
        }
        // normals
        if (Properties.Settings.Default.RenderNormals)
        {
            if (_vbo.NormalBufferId != 0)
            {
                GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.NormalBufferId);
                GL.NormalPointer(NormalPointerType.Float, Vector3.SizeInBytes, IntPtr.Zero);
                GL.EnableClientState(ArrayCap.NormalArray);
            }
        }
        // vertex colors
        if (Properties.Settings.Default.RenderVertexColors)
        {
            GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.ColorBufferId);
            GL.ColorPointer(4, ColorPointerType.UnsignedByte, sizeof(int), IntPtr.Zero);
            GL.EnableClientState(ArrayCap.ColorArray);
        }
        // UV coordinates
        if (Properties.Settings.Default.RenderTexture)
        {
            if (_vbo.TexCoordBufferId != 0)
            {
                GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.TexCoordBufferId);
                GL.TexCoordPointer(2, TexCoordPointerType.Float, 8, IntPtr.Zero);
                GL.EnableClientState(ArrayCap.TextureCoordArray);
            }
        }
        // vertex position
        GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.VertexBufferId);
        GL.VertexPointer(3, VertexPointerType.Float, Vector3.SizeInBytes, IntPtr.Zero);
        GL.EnableClientState(ArrayCap.VertexArray);
        // primitives
        GL.BindBuffer(BufferTarget.ElementArrayBuffer, _vbo.ElementBufferId);
        GL.DrawElements(BeginMode.Triangles, _vbo.NumIndices /* actually, count(indices) */,
            DrawElementsType.UnsignedShort, IntPtr.Zero);
        // Restore the state
        GL.PopClientAttrib();
    }

    bool _buffer_mapped = false;

    /// Call this to get a pointer to OpenGL private memory buffer.
    public void BeginModifyVertexData(out IntPtr data, out int qty_vertices)
    {
        Debug.Assert(_buffer_mapped == false, "Forgot to unmap the buffer with GL.UnmapBuffer()");
        _buffer_mapped = true;
        GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.VertexBufferId);
        data = GL.MapBuffer(BufferTarget.ArrayBuffer, BufferAccess.ReadWrite);
        // note: number of floats in "data" = (qty_vertices * 3)
        qty_vertices = _mesh.VertexCount;
```

```
}
/// Call this when done working with OpenGL memory. This uploads it back into OpenGL.
public void EndModifyVertexData()
{
    bool data_upload_ok = GL.UnmapBuffer(BufferTarget.ArrayBuffer);
    if (! data_upload_ok)
    {
        // data store contents have become corrupt while the data store was mapped
        // This can occur for system-specific reasons that affect the availability
        // of graphics memory, such as screen mode changes.
        // Then GL_FALSE is returned and data contents are undefined
        // An application must detect this rare condition and reinitialize the data store.
        // We will not reinitialise the store, but simply bail out.
        throw new Exception("OpenGL driver has failed.");
    }
    _buffer_mapped = false;
}

public void BeginModifyNormalData(out IntPtr data, out int qty_normals)
{
    Debug.Assert(_buffer_mapped == false, "Forgot to unmap the buffer with GL.UnmapBuffer()");
    _buffer_mapped = true;
    GL.BindBuffer(BufferTarget.ArrayBuffer, _vbo.NormalBufferId);
    data = GL.MapBuffer(BufferTarget.ArrayBuffer, BufferAccess.ReadWrite);
    // note: number of floats in "data" = (qty_normals * 3)
    qty_normals = _mesh.Normals.Count;
}
public void EndModifyNormalData()
{
    bool data_upload_ok = GL.UnmapBuffer(BufferTarget.ArrayBuffer);
    if (! data_upload_ok)
    {
        // data store contents have become corrupt while the data store was mapped
        // This can occur for system-specific reasons that affect the availability
        // of graphics memory, such as screen mode changes.
        // Then GL_FALSE is returned and data contents are undefined
        // An application must detect this rare condition and reinitialize the data store.
        // We will not reinitialise the store, but simply bail out.
        throw new Exception("OpenGL driver has failed.");
    }
    _buffer_mapped = false;
}

public int CompileMaterialDisplayList()
{
    int id = GL.GenLists(1);
    GL.NewList(id, ListMode.Compile);
    ApplyMaterial();
    GL.EndList();
    return id;
}

public OpenTK.Graphics.Color4 Assimp2OpenTK(Assimp.Color4D input)
{
    return new Color4(input.R, input.G, input.B, input.A);
}

double AlphaSuppressionThreshold = 0.01;

/// Apply material properties to the model.
private void ApplyMaterial()
{
    var hasColors = _mesh != null && _mesh.HasVertexColors(0);
    if (hasColors)
    {
        GL.Enable(EnableCap.ColorMaterial);
        GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);
    }
    else
    {
        GL.Disable(EnableCap.ColorMaterial);
    }
    // note: keep semantics of hasAlpha consistent with IsAlphaMaterial()
    var hasAlpha = false;
    var hasTexture = false;

    GL.Disable(EnableCap.Texture2D);
    GL.Enable(EnableCap.Normalize);

    var alpha = 1.0f;
    if (_material.HasOpacity)
    {
```

```
    alpha = _material.Opacity;
    // ignore zero alpha channel
    if (alpha < AlphaSuppressionThreshold)
    {
        alpha = 1.0f;
    }
}

var color = new Color4(.8f, .8f, .8f, 1.0f);
if (_material.HasColorDiffuse)
{
    color = Assimp2OpenTK(_material.ColorDiffuse);
    if (color.A < AlphaSuppressionThreshold) // s.a.
    {
        color.A = 1.0f;
    }
}
color.A *= alpha;
hasAlpha = hasAlpha || color.A < 1.0f;

// if the material has a texture but the diffuse color texture is all black,
// then heuristically assume that this is an import/export flaw and substitute
// white.
if (hasTexture && color.R < 1e-3f && color.G < 1e-3f && color.B < 1e-3f)
{
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, Color4.White);
}
else
{
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, color);
}

color = new Color4(0, 0, 0, 1.0f);
if (_material.HasColorSpecular)
{
    color = Assimp2OpenTK(_material.ColorSpecular);
}
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, color);

color = new Color4(.2f, .2f, .2f, 1.0f);
if (_material.HasColorAmbient)
{
    color = Assimp2OpenTK(_material.ColorAmbient);
}
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, color);

color = new Color4(0, 0, 0, 1.0f);
if (_material.HasColorEmissive)
{
    color = Assimp2OpenTK(_material.ColorEmissive);
}
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Emission, color);

float shininess = 1;
float strength = 1;
if (_material.HasShininess)
{
    shininess = _material.Shininess;

}
// todo: I don’t even remember how shininess strength was supposed to be handled in assimp
if (_material.HasShininessStrength)
{
    strength = _material.ShininessStrength;
}

var exp = shininess*strength;
if (exp >= 128.0f) // 128 is the maximum exponent as per the Gl spec
{
    exp = 128.0f;
}

GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, exp);

if (hasAlpha)
{
    GL.Enable(EnableCap.Blend);
    GL.BlendFunc(BlendingFactorSrc.SrcAlpha, BlendingFactorDest.OneMinusSrcAlpha);
    GL.DepthMask(false);
}
else
{
        GL.Disable(EnableCap.Blend);
        GL.DepthMask(true);
    }
}


/// <summary>
/// Currently only called during construction, this method uploads the input mesh (
/// the RenderMesh instance is bound to) to a VBO.
/// </summary>
/// <param name="vboToFill"></param>
private void Upload(out Vbo vboToFill)
{
    vboToFill = new Vbo();
    UploadVertices(out vboToFill.VertexBufferId);
    if (_mesh.HasNormals)
    {
        UploadNormals(out vboToFill.NormalBufferId);
    }
    //if (_mesh.HasVertexColors(0))
    //{
    //    UploadColors(out vboToFill.ColorBufferId);
    //}
    //if (_mesh.HasTextureCoords(0))
    //{
    //    UploadTextureCoords(out vboToFill.TexCoordBufferId);
    //}
    UploadPrimitives(out vboToFill.ElementBufferId, out vboToFill.NumIndices);
    // TODO: upload bone weights
}

/// <summary>
/// Generates and populates an Gl vertex array buffer given 3D vectors as source data
/// </summary>
private void NewServerBufferWithFloats(out int outGlBufferId, List<Vector3D> dataBuffer)
{
    GL.GenBuffers(1, out outGlBufferId);
    GL.BindBuffer(BufferTarget.ArrayBuffer, outGlBufferId);
    int sizeof_vec3d = 12; // X,Y,Z = 3 floats, 4 bytes each
    var byteCount = dataBuffer.Count * sizeof_vec3d;
    var temp = new float[byteCount];
    var n = 0;
    foreach(var v in dataBuffer)
    {
        temp[n++] = v.X;
        temp[n++] = v.Y;
        temp[n++] = v.Z;
    }
    GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)byteCount, temp, BufferUsageHint.StreamDraw);
    VerifyArrayBufferSize(byteCount);
    GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
}


/// <summary>
/// Verifies that the size of the currently bound vertex array buffer matches
/// a given parameter and throws if it doesn't.
/// </summary>
private void VerifyArrayBufferSize(int byteCount)
{
    int bufferSize;
    GL.GetBufferParameter(BufferTarget.ArrayBuffer, BufferParameterName.BufferSize, out bufferSize);
    if (byteCount != bufferSize)
    {
        throw new Exception("Vertex data array not uploaded correctly - buffer size does not match upload size");
    }
}


/// <summary>
/// Uploads vertex indices to a newly generated Gl vertex array
/// </summary>
private void UploadPrimitives(out int elementBufferId, out int indicesCount)
{
    //Debug.Assert(_mesh.HasTextureCoords(0));

    GL.GenBuffers(1, out elementBufferId);
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, elementBufferId);

    var faces = _mesh.Faces;

    // TODO account for other primitives than triangles
    var triCount = 0;
    int byteCount;
    foreach(var face in faces)
    {
        Debug.Assert(face.IndexCount == 3);
        ++triCount;
    }
    var intCount = triCount * 3;

    // since we are 64 bit compile target
    var temp = new ushort[intCount];
    byteCount = intCount * sizeof(ushort);
    var n = 0;
    foreach (var idx in faces.Where(face => face.IndexCount == 3).SelectMany(face => face.Indices))
    {
        Debug.Assert(idx <= 0xffff);
        temp[n++] = (ushort)idx;
    }
    GL.BufferData(BufferTarget.ElementArrayBuffer, (IntPtr)byteCount, temp, BufferUsageHint.StaticDraw);


    int bufferSize;
    GL.GetBufferParameter(BufferTarget.ElementArrayBuffer, BufferParameterName.BufferSize, out bufferSize);
    if (byteCount != bufferSize)
    {
        throw new Exception("Index data array not uploaded correctly - buffer size does not match upload size");
    }

    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);
    indicesCount = triCount * 3;
}


/// <summary>
/// Uploads UV coordinates to a newly generated Gl vertex array.
/// </summary>
private void UploadTextureCoords(out int texCoordBufferId)
{
    Debug.Assert(_mesh.HasTextureCoords(0));

    GL.GenBuffers(1, out texCoordBufferId);
    GL.BindBuffer(BufferTarget.ArrayBuffer, texCoordBufferId);

    var uvs = _mesh.TextureCoordinateChannels[0];
    var floatCount = uvs.Count * 2;
    var temp = new float[floatCount];
    var n = 0;
    foreach (var uv in uvs)
    {
        temp[n++] = uv.X;
        temp[n++] = uv.Y;
    }

    var byteCount = floatCount*sizeof (float);
    GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(byteCount), temp, BufferUsageHint.StaticDraw);
    VerifyArrayBufferSize(byteCount);
    GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
}


/// <summary>
/// Uploads vertex positions to a newly generated Gl vertex array.
/// </summary>
private void UploadVertices(out int verticesBufferId)
{
    NewServerBufferWithFloats(out verticesBufferId, _mesh.Vertices);
}


/// <summary>
/// Uploads normal vectors to a newly generated Gl vertex array.
/// </summary>
private void UploadNormals(out int normalBufferId)
{
    Debug.Assert(_mesh.HasNormals);
    NewServerBufferWithFloats(out normalBufferId, _mesh.Normals);
}


/// <summary>
/// Uploads tangents and bitangents to newly generated Gl vertex arrays.
/// </summary>
```

```
        private void UploadTangentsAndBitangents(out int tangentBufferId, out int bitangentBufferId)
        {
            Debug.Assert(_mesh.HasTangentBasis);

            var tangents = _mesh.Tangents;
            NewServerBufferWithFloats(out tangentBufferId, tangents);

            var bitangents = _mesh.BiTangents;
            Debug.Assert(bitangents.Count == tangents.Count);

            NewServerBufferWithFloats(out bitangentBufferId, bitangents);
        }


        /// <summary>
        /// Uploads vertex colors to a newly generated Gl vertex array.
        /// </summary>
        /// <param name="colorBufferId"></param>
        private void UploadColors(out int colorBufferId)
        {
            Debug.Assert(_mesh.HasVertexColors(0));

            GL.GenBuffers(1, out colorBufferId);
            GL.BindBuffer(BufferTarget.ArrayBuffer, colorBufferId);

            var colors = _mesh.VertexColorChannels[0];
            // convert to 32Bit RGBA
            var byteCount = colors.Count*4;
            var byteColors = new byte[byteCount];
            var n = 0;
            foreach(var c in colors)
            {
                byteColors[n++] = (byte)(c.R * 255);
                byteColors[n++] = (byte)(c.G * 255);
                byteColors[n++] = (byte)(c.B * 255);
                byteColors[n++] = (byte)(c.A * 255);
            }

            GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(byteCount), byteColors, BufferUsageHint.StaticDraw);
            VerifyArrayBufferSize(byteCount);
            GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Drawing;

namespace WinFormAnimation2D
{
    /// <summary>
    /// Simple class to store mouse status data.
    /// Monitor mouse status (delta, position, click_position, etc.)
    /// </summary>
    class MouseState
    {

        // Is the mouse being pressed down currently.
        public bool IsPressed;

        /// Current mouse position, this is updated by you.
        public Point CurrentPos;
        /// Captured mouse position when it was clicked.
        public Point ClickPos;

        public Point LastFramePos;
        public Point FrameDelta
        {
            get
            {
                return new Point(LastFramePos.X - CurrentPos.X, LastFramePos.Y - CurrentPos.Y);
            }
        }

        /// Position of where the user is pointing inside the game world
        public OpenTK.Vector3 InnerWorldPos;
        /// Position of click inside the game world.
        public OpenTK.Vector3 InnerWorldClickPos;

        /// Mimimum motion delta for mouse to be recognised
        public readonly int HorizHysteresis = 4;
        public readonly int VertHysteresis = 4;

        /// Updates mouse click position.
        public void RecordMouseClick(MouseEventArgs e)
        {
            this.ClickPos = new Point(e.X, e.Y);
            this.IsPressed = true;
        }

        /// Updates current mouse position. Then we can caluclate delta better.
        public void RecordMouseMove(MouseEventArgs e)
        {
            this.LastFramePos = this.CurrentPos;
            this.CurrentPos = new Point(e.X, e.Y);
        }

        public void RecordInnerWorldMouseClick(OpenTK.Vector3 vec)
        {
            this.InnerWorldClickPos = vec;
            this.InnerWorldPos = vec;
        }

        public void RecordInnerWorldMouseMove(OpenTK.Vector3 vec)
        {
            this.InnerWorldPos = vec;
        }

    };
}

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 OpentkMatrixExtensions
    {

        /// <summary>
        /// Convert 4x4 OpenTK matrix into Assimp matrix. This should not be used often.
        /// </summary>
        /// <param name="m"></param>
        /// <returns></returns>
        public static ai.Matrix4x4 eToAssimp(this tk.Matrix4 m)
        {
            return new ai.Matrix4x4
            {
               A1 = m.M11,
               B1 = m.M12,
               C1 = m.M13,
               D1 = m.M14,
               A2 = m.M21,
               B2 = m.M22,
               C2 = m.M23,
               D2 = m.M24,
               A3 = m.M31,
               B3 = m.M32,
               C3 = m.M33,
               D3 = m.M34,
               A4 = m.M41,
               B4 = m.M42,
               C4 = m.M43,
               D4 = m.M44
            };
        }

        /// <summary>
        /// Convert _OpenTK_ 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 tk.Matrix4 m)
        {
            return m.eToAssimp().eTo3x2();
        }
    }
}

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 OpentkVectorExtensions
    {

        /// <summary>
        /// Convert _OpenTK_ 3D vector to 2D System.Drawing.Point
        /// for drawing with Graphics object.
        /// </summary>
        public static Point eToPoint(this tk.Vector3 v)
        {
            return new Point((int)v.X, (int)v.Y);
        }

        /// <summary>
        /// Convert _OpenTK_ 3D vector to 2D System.Drawing.PointF (floating point)
        /// for drawing with Graphics object.
        /// </summary>
        public static PointF eToPointFloat(this tk.Vector3 v)
        {
            return new PointF(v.X, v.Y);
        }

        /// <summary>
        /// Convert _OpenTK_ 2D vector to 2D System.Drawing.Point
        /// for drawing with Graphics object.
        /// </summary>
        public static Point eToPoint(this tk.Vector2 v)
        {
            return new Point((int)v.X, (int)v.Y);
        }

        /// <summary>
        /// Convert _OpenTK_ 2D vector to 2D System.Drawing.PointF (floating point)
        /// for drawing with Graphics object.
        /// </summary>
        public static PointF eToPointFloat(this tk.Vector2 v)
        {
            return new PointF(v.X, v.Y);
        }

        /// <summary>
        /// Convert open tk 3D vector to opentk 2D vector.
        /// </summary>
        public static tk.Vector2 eTo2D(this tk.Vector3 v)
        {
            return new tk.Vector2(v.X, v.Y);
        }

        /// <summary>
        /// Checks if the vector has all values close to zero.
        /// </summary>
        public static bool eIsZero(this tk.Vector3 v)
        {
            if (Math.Abs(v.X) < Util.epsilon
                && Math.Abs(v.Y) < Util.epsilon
                && Math.Abs(v.Z) < Util.epsilon)
            {
                return true;
            }
            return false;
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WinFormAnimation2D
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
using Microsoft.CSharp;
using System.CodeDom.Compiler;
using System.Reflection;

namespace WinFormAnimation2D
{
    class RecentFilesFolders
    {
        private string _currently_open_filepath;
        public string CurrentlyOpenFilePath
        {
            get { return _currently_open_filepath; }
            set
            {
                _currently_open_filepath = value;
                CurrentlyOpenFilePathChanged(_currently_open_filepath);
            }
        }

        /// <summary>
        /// Event to fire on current filepath change.
        /// </summary>
        public event Action<string> CurrentlyOpenFilePathChanged;


        public RecentFilesFolders()
        {
            if (Properties.Settings.Default.RecentFiles == null)
            {
                Properties.Settings.Default.RecentFiles = new StringCollection();
                Properties.Settings.Default.Save();
            }
            /// Set sensible value for directory in OpenFile dialog.
            if (Properties.Settings.Default.RecentDirectory == null)
            {
                Properties.Settings.Default.RecentDirectory
                    = Environment.ExpandEnvironmentVariables("%HOMEDRIVE%%HOMEPATH%");
            }
        }

        /// <summary>
        /// The OpenRecent files have changed. Refresh the view in menu.
        /// </summary>
        public void ReplaceOpenRecentMenu(ToolStripMenuItem open_recent_menu, Action<string> onclick)
        {
            open_recent_menu.DropDownItems.Clear();
            if (Properties.Settings.Default.RecentFiles.Count == 0)
            {
                open_recent_menu.DropDownItems.Add("No recent files...");
                return;
            }
            foreach (var str in Properties.Settings.Default.RecentFiles)
            {
                var tool = open_recent_menu.DropDownItems.Add(str);
                // so that "str" var is not cached by LINQ use "tool.Text"
                tool.Click += delegate { onclick(tool.Text); };
            }
        }

        /// <summary>
        /// Add a new item to the Recent-Files menu and save it persistently
        /// </summary>
        /// <param name="file"></param>
        public void AddRecentFile(string file)
        {
            var recent = Properties.Settings.Default.RecentFiles;
            if (recent.Count > Properties.Settings.Default.QtyOfRecentFiles)
            {
                recent.RemoveAt(recent.Count - 1);
            }
            // Reinsert at 0 position (does not throw if not found)
            recent.Remove(file);
            recent.Insert(0, file);
            Properties.Settings.Default.Save();
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing.Drawing2D;
using System.Drawing;
using System.Windows.Forms;
using OpenTK.Graphics.OpenGL;
using OpenTK;

namespace WinFormAnimation2D
{
    /// <summary>
    /// Class to control openGL settings and do the actual drawing.
    /// All openGL calls will be here.
    /// </summary>
    class Renderer
    {
        // The control to which we are rendering to.
        // Change this to OpenGL control later.
        private PictureBox _canvas;

        public DrawConfig GlobalDrawConf;

        public Renderer()
        {
        }

        public void ClearFrameBuffer()
        {
        }

        /// Enable default OpenGL settings. Set lights, material, etc. Call this once in the beginning.
        public void InitOpenGL()
        {
            // enable stuff
            GL.Enable(EnableCap.ColorMaterial);
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.Lighting);
            // other settings
            GL.ShadeModel(ShadingModel.Flat);
            GL.ClearColor(Color.DarkGray);
            GL.Hint(HintTarget.PerspectiveCorrectionHint, HintMode.Nicest);
            // lights
            GL.Enable(EnableCap.Light0);
            GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0, 0, 10, 0 });
        }
```

```
/// Called when window size changes.
public void ResizeOpenGL(int width, int height)
{
    GL.Viewport(0, 0, width, height);
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    // set a proper perspective matrix for rendering
    float aspectRatio = ((float)width)/height;
    Matrix4 perspective = Matrix4.CreatePerspectiveFieldOfView(MathHelper.PiOver4, aspectRatio, 0.1f, 1000.0f);
    GL.LoadMatrix(ref perspective);
    // now Model view matrix
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
}

/// <summary>
/// Important points to remember:
/// Set normals.
/// Must be clock wise vertex draw order
/// The x-axis is accross the screen, so the Z-axis triangle must have component along X: +-1
/// since look at looks towards the center, we need to offset it a bit to see the Z axis.
/// </summary>
public void DrawAxis3D()
{
    GL.Disable(EnableCap.DepthTest);
    GL.Enable(EnableCap.ColorMaterial);
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.AmbientAndDiffuse, Color.Aqua);
    GL.Normal3(0, 0, 1);
    int shift = 1;
    GL.Begin(BeginMode.Triangles);
    // x axis
    GL.Color3(1.0f, 0.0f, 0.0f);
    GL.Vertex3(0, 0, 0);
    GL.Vertex3(20, -shift, 0);
    GL.Vertex3(20, shift, 0);
    // y axis
    GL.Color3(0.0f, 1.0f, 0.0f);
    GL.Vertex3(0, 0, 0);
    GL.Vertex3(shift, 20, 0);
    GL.Vertex3(-shift, 20, 0);
    // z axis
    GL.Color3(0.0f, 0.0f, 1.0f);
    GL.Vertex3(0, 0, 0);
    GL.Vertex3(-shift, 0, 20);
    GL.Vertex3(shift, 0, 20);
    GL.End();
    GL.Enable(EnableCap.DepthTest);
}

/// Prepare to render next OpenGL frame. Clear depth/color buffers.
public void ClearOpenglFrameForRender(Matrix4 camera_matrix)
{
    // TEST CODE to visualize mid point (pivot) and origin
    // var view = Matrix4.LookAt(0, 50, 500, 0, 0, 0, 0, 1, 0);
    //GL.LoadMatrix(ref view);
    if (Properties.Settings.Default.OpenGLCullFace)
    {
        GL.Enable(EnableCap.CullFace);
    }
    else
    {
        GL.Disable(EnableCap.CullFace);
    }
    GL.LoadIdentity();
    GL.LoadMatrix(ref camera_matrix);
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    // light color
    var col = new Vector3(1, 1, 1);
    col *= (0.25f + 1.5f * 10 / 100.0f) * 1.5f;
    GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { col.X, col.Y, col.Z, 1 });
    // 3d
    GL.Enable(EnableCap.DepthTest);
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
}

public void DrawEmptyEntitySplash()
{
    string msg = "No file loaded";
    //var w = (float)RenderResolution.Width;
    //var h = (float)RenderResolution.Height;
    // Util.GR.DrawString(msg, GlobalDrawConf.DefaultFont16 , Brushes.Aquamarine, new PointF(w / 2.0f, h / 2.0f));
```

```
        }

    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Linq;
using Assimp;
using OpenTK;
using Quaternion = Assimp.Quaternion;

namespace WinFormAnimation2D
{
    class SceneWrapper
    {
        public Scene _inner;
        // collada file stores armature as a separate node
        public Dictionary<string,BoneNode> _name2bone_node = new Dictionary<string,BoneNode>();
        public Dictionary<string,Node> _name2node = new Dictionary<string, Node>();

        public SceneWrapper(Scene sc)
        {
            _inner = sc;
            InnerBuildNodeDict(sc.RootNode);
        }

        public void InnerBuildNodeDict(Node nd)
        {
            _name2node[nd.Name] = nd;
            foreach (var child in nd.Children)
            {
                InnerBuildNodeDict(child);
            }
        }

        public Node GetNode(string name)
        {
            return _name2node[name];
        }

        public BoneNode GetBoneNode(string node_name)
        {
            return _name2bone_node[node_name];
        }

        public BoneNode BuildBoneNodes(string armature_root_name)
        {
            Node armature_root = InnerRecurFindNode(_inner.RootNode, armature_root_name);
            BoneNode root = InnerRecurBuildBones(armature_root);
            return root;
        }

        private BoneNode InnerRecurBuildBones(Node nd)
        {
            var current = new BoneNode(nd);
            current.GlobTrans = GetNodeGlobalTransform(nd);
            current.LocTrans = nd.Transform;
            // add to dict for faster lookup
            _name2bone_node[nd.Name] = current;
            foreach (var child in nd.Children)
            {
                BoneNode w_child = InnerRecurBuildBones(child);
                w_child.Parent = current;
                current.Children.Add(w_child);
            }
            return current;
        }

        /// <summary>
        /// Get the bone transform and trace back its changes
        /// </summary>
        /// <param name="nd"></param>
        /// <returns></returns>
        public Matrix4x4 GetNodeGlobalTransform(Node nd)
        {
            Matrix4x4 ret = new Matrix4x4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);
            ret *= nd.Transform;
            Node cur = nd;
            while (cur.Parent != null)
            {
                ret *= cur.Parent.Transform;
                cur = cur.Parent;
            }
            return ret;
        }

        // Use only NodeWrapper in the interface to outside
        // Deprecated use GetNode
        public Node FindNode(string node_name)
        {
            return InnerRecurFindNode(_inner.RootNode, node_name);
        }

        private Node InnerRecurFindNode(Node cur_node, string node_name)
        {
            if (cur_node.Name == node_name)
            {
                return cur_node;
            }
            foreach (var child in cur_node.Children)
            {
                var tmp =  InnerRecurFindNode(child, node_name);
                if (tmp != null)
                {
                    return tmp;
                }
            }
            return null;
        }

        /// <summary>
        /// Make sure that all meshes are named.
        /// </summary>
        public void NameUnnamedMeshes()
        {
            for (int i = 0; i < _inner.MeshCount; i++)
            {
                Mesh mesh = _inner.Meshes[i];
                if (mesh.Name.Length == 0)
                {
                    mesh.Name = i.ToString();
                }
            }
        }

        public void NodeNamesAreUnique()
        {
            var name_set = new HashSet<string>();
            InnerRecurCheckNamesUnique(_inner.RootNode, name_set);
        }

        public void InnerRecurCheckNamesUnique(Node nd, HashSet<string> nd_names)
        {
            if (nd_names.Contains(nd.Name))
            {
                throw new Exception("Node names in scene are not unique. Can not proceed.");
            }
            else
            {
                nd_names.Add(nd.Name);
            }
            foreach (var child in nd.Children)
            {
                InnerRecurCheckNamesUnique(child, nd_names);
            }
        }

    }
}

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>
    /// Implement this when class allows local matrix transforms.
    /// (Entity, Camera)
    /// </summary>
    interface ITransformState
    {
        TransformState Transform { get; }
        /// Get the translation part of the matrix
        Vector3 GetTranslation { get; }

        /// Rotate by angle around default axis. Called on mouse events.
        void RotateBy(double angle_degrees);

        /// x,y,z should be direction parameters, one of {-1, 0, 1}. Called on keyboard events.
        void MoveBy(Vector3 direction);
    }

    class TransformState
    {
        public float MoveSpeed;
        public float RotateSpeedDegrees;

        public Matrix4 _matrix = Matrix4.Identity;
        public Matrix4x4 _ai_matrix
        {
            get { return _matrix.eToAssimp(); }
            set { _matrix = value.eToOpenTK(); }
        }

        public TransformState(Matrix4 init_matrix, double motion_speed, double rotation_speed_degrees)
        {
            _matrix = init_matrix;
            MoveSpeed = (float)motion_speed;
            RotateSpeedDegrees = (float)rotation_speed_degrees;
        }

        public Vector3 GetTranslation
        {
            get { return _matrix.ExtractTranslation(); }
        }
        public Vector2 GetTranslation2D
        {
            get { return _matrix.ExtractTranslation().eTo2D(); }
        }

        public void Rotate(double angle_degrees)
        {
            // we must use global vectors here, becasue we pre-multiply the camera matrix and then invert it
            RotateAroundAxis(angle_degrees, Vector3.UnitX);
        }
        public void RotateAroundAxis(double angle_degrees, Vector3 axis)
        {
            float angle_radians = (float)(angle_degrees * Math.PI / 180.0);
            _matrix = Matrix4.CreateFromAxisAngle(axis, angle_radians) * _matrix;
        }

        // x,y,z should be direction parameters, one of {-1, 0, 1}
        public Vector3 TranslationFromDirectionInPlaneYZ(Vector2 direction)
        {
            // becasue we move perpendicular to camera direction, i.e. perpendicular to camera's X axis
            Vector3 _local_y = _matrix.Row1.Xyz;
            Vector3 _local_z = _matrix.Row2.Xyz;
            Vector3 dir = direction.X * _local_y + direction.Y * _local_z;
            return Vector3.Multiply(dir, MoveSpeed);
        }

        // x,y,z should be direction parameters, one of {-1, 0, 1}
        public Vector3 TranslationFromDirection(Vector3 direction)
        {
            Debug.Assert(direction.Length > 0);
            direction.Normalize();
            return Vector3.Multiply(direction, MoveSpeed);
        }

        public void ApplyTranslation(Vector3 trans)
        {
            _matrix = Matrix4.CreateTranslation(trans) * _matrix;
        }

        public void MoveBy(Vector3 direction)
        {
            Vector3 trans = TranslationFromDirection(direction);
            ApplyTranslation(trans);
        }

    }

}

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;

namespace WinFormAnimation2D
{
    public enum TreeNodeType
    {
        Entity = 0
        , Mesh
        , TriangleFace
        , Armature
        , Other
        ,
    }

    interface IHighlightableNode
    {
        void Render();
    }

    class SceneTreeNode : TreeNode, IHighlightableNode
    {
        public TreeNodeType NodeType = TreeNodeType.Other;

        public void Render()
        {
        }

        public SceneTreeNode(string name)
        {
            this.Name = name;
            this.Text = name;
        }
    }

    class EntityTreeNode : TreeNode, IHighlightableNode
    {
        public TreeNodeType NodeType = TreeNodeType.Entity;
        public BoundingBoxGroup DrawMeshBounds;

        public void Render()
        {
            DrawMeshBounds.OverallBox.Render();
        }
        public EntityTreeNode(string name)
        {
            this.Name = name;
            this.Text = name;
        }
    }
    class MeshTreeNode : TreeNode, IHighlightableNode
    {
        public TreeNodeType NodeType = TreeNodeType.Mesh;
        public BoundingBoxGroup DrawData;

        public void Render()
        {
            DrawData.OverallBox.Render();
        }

        public MeshTreeNode(string name)
        {
            this.Name = name;
            this.Text = name;
        }
    }

    class ArmatureTreeNode : TreeNode, IHighlightableNode
    {
        public TreeNodeType NodeType = TreeNodeType.Armature;
        public BoneBounds DrawData;

        public void Render()
        {
            DrawData.Render();
        }
        public ArmatureTreeNode(string name)
        {
            this.Name = name;
            this.Text = name;
        }
    }

}

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;
using System.Reflection;

namespace WinFormAnimation2D
{

    public static class Breakpoints
    {
        public static bool Allow = false;
    }

    public static class Util
    {

        // Useful for random value generation.
        // We use only one Random instance in the whole program.
        private static Random rand = new Random();

        // Note that this is Unsigned int (so overflow is ok)
        public static Func<Brush> GetNextBrush = SetupBrushGen();
        public static Func<Color> GetNextColor = SetupColorGen();

        // Static config fields
        public static double epsilon = 1E-8;

        /// <summary>
        /// Big + Green pen to render points on screen
        /// </summary>
        public static Pen pp1 = new Pen(Color.LawnGreen, 20.0f);
        public static Color cc1 = Color.LawnGreen;

        /// <summary>
        /// Medium + Black pen to render points on screen
        /// </summary>
        public static Pen pp2 = new Pen(Color.Black, 10.0f);
        public static Color cc2 = Color.Black;
```

```
/// <summary>
/// Small + Red pen to render points on screen
/// </summary>
public static Pen pp3 = new Pen(Color.Red, 0.01f);
public static Color cc3 = Color.Red;

/// <summary>
/// Small + Red pen to render points on screen
/// </summary>
public static Pen pp4 = new Pen(Color.SkyBlue, 2.5f);
public static Color cc4 = Color.SkyBlue;

/// <summary>
/// Get a brush of next color. (to distingush rendered polygons)
/// </summary>
/// <returns></returns>
private static Func<Brush> SetupBrushGen()
{
   // Note that this is Unsigned int (so overflow is ok)
   uint _iter_nextbrush = 0;
   return () =>
   {
      if (Properties.Settings.Default.TriangulateMesh == false)
      {
         return Brushes.Green;
      }
      // cache this variable
      _iter_nextbrush++;
      switch (_iter_nextbrush % 3)
      {
         case 0:
            return Brushes.GreenYellow;
         case 1:
            return Brushes.SeaGreen;
         case 2:
            return Brushes.Green;
         case 3:
            return Brushes.LightSeaGreen;
         case 4:
            return Brushes.LawnGreen;
         default:
            return Brushes.Red;
      }
   };
}

/// <summary>
/// Get a brush of next color. (to distingush rendered polygons)
/// </summary>
/// <returns></returns>
private static Func<Color> SetupColorGen()
{
   // Note that this is Unsigned int (so overflow is ok)
   uint _iter_next_color = 0;
   return () =>
   {
      if (Properties.Settings.Default.TriangulateMesh == false)
      {
         return Color.Green;
      }
      // cache this variable
      _iter_next_color++;
      switch (_iter_next_color % 3)
      {
         case 0:
            return Color.GreenYellow;
         case 1:
            return Color.SeaGreen;
         case 2:
            return Color.Green;
         case 3:
            return Color.LightSeaGreen;
         case 4:
            return Color.LawnGreen;
         default:
            return Color.Red;
      }
   };
}

// subtract one point from another
public static Point Minus(this Point a, Point b)
```

```
        {
            return new Point(a.X - b.X, a.Y - b.Y);
        }
        // add one point to another
        public static Point Add(this Point a, Point b)
        {
            return new Point(a.X + b.X, a.Y + b.Y);
        }

    } // end of class
}

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 d2d = System.Drawing.Drawing2D;
using OpenTK;
using System.Diagnostics;


namespace WinFormAnimation2D
{
    class World
    {
        public Logger _logger = new Logger("skeletal_animation.txt");
        public Entity _enttity_one = null;
        public Renderer _renderer = null;

        public bool HasScene = false;

        public SceneWrapper _cur_scene;
        public NodeInterpolator _action_one;

        private Entity _currently_selected;
        public Entity CurrentlySelected
        {
            get { return _currently_selected; }
            private set { _currently_selected = value; }
        }

        public World()
        {
            _renderer = new Renderer();
            _renderer.GlobalDrawConf = new DrawConfig
            {
                EnablePolygonModeFill = true,
                EnableLight = true,
            };
        }

        public void LoadScene(byte[] filedata)
        {
            MemoryStream sphere = new MemoryStream(filedata);
            var assimp_scene = BuildAssimpScene(sphere, "dae");
            _cur_scene = new SceneWrapper(assimp_scene);
            _cur_scene.NameUnnamedMeshes();
            _cur_scene.NodeNamesAreUnique();
            // load other data
            _action_one = new NodeInterpolator(_cur_scene, _cur_scene._inner.Animations[0]);
            BoneNode armature = _cur_scene.BuildBoneNodes("Armature");
            string mesh_default_name = "Cube";
            Node mesh = _cur_scene.FindNode(mesh_default_name);
            if (mesh == null)
            {
                throw new Exception("Could not find node named " + mesh_default_name);
            }
            ActionState state = new ActionState(_cur_scene._inner.Animations[0]);
            _enttity_one = new Entity(_cur_scene, mesh, armature, state);
            state._owner = _enttity_one;
            _action_one.ApplyAnimation(_enttity_one._armature, _enttity_one._action);
            HasScene = true;
        }

        public Scene BuildAssimpScene(MemoryStream model_data, string format_hint)
        {
            Scene tmp_scene;
            using (var importer = new AssimpContext())
            {
                importer.SetConfig(new NormalSmoothingAngleConfig(66.0f));
                LogStream logstream = new LogStream((msg, userData) => _logger.Log(msg));
                logstream.Attach();
                tmp_scene = importer.ImportFileFromStream(model_data
                    , PostProcessPreset.TargetRealTimeFast
                    , format_hint);
                // we could load the model into our own data structures here
            }
            if (tmp_scene == null || tmp_scene.SceneFlags.HasFlag(SceneFlags.Incomplete))
            {
                throw new Exception("Bad file format. Could not read data.");
            }
            return tmp_scene;
        }

        // dt = delta time since last frame in milliseconds
        public void Update(double dt_millisecs)
        {
            if (HasScene)
            {
                _enttity_one.UpdateModel(dt_millisecs);
            }
        }

        /// <summary>
        /// Render the model stored in EntityScene useing the Graphics object.
        /// </summary>
        public void RenderWorld()
        {
            if (HasScene)
            {
                _enttity_one.RenderModel(_renderer.GlobalDrawConf);
            }
        }

        public bool CheckMouseEntitySelect(MouseState mouse_state)
        {
            if (! HasScene)
            {
                return false;
            }
            var vec = new Vector2(mouse_state.InnerWorldPos.X, mouse_state.InnerWorldPos.Y);
            if (_enttity_one.ContainsPoint(vec))
            {
                CurrentlySelected = _enttity_one;
                return true;
            }
            return false;
        }

    }
}
```

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					