**What is K3D.js?**

K3D is JavaScript library for **working with 3D meshes**. K3D is **not a rendering engine**!

K3D.js can be used for different operations with 3D meshes:

* converting between different formats
* applying affine and other transformations
* generating primitive objects
* creating LODs

To use K3D, you should have a basic knowledge of 3D meshes, index / vertex coordinate arrays etc.

**Documentation**

There are no "classes", constructors or special data types in K3D.js. The library consists of static functions only. It is very fast.

**JSONable** object is a Javascript object, that can be converted to and from String using JSON.stringify and JSON.parse.

* **load** ( url : String, resp : Function )

Loads any file, which is handled to "resp" as ArrayBuffer after loading.

* **save** ( buff : ArrayBuffer, name : String )

Saves any file into client's computer.

* **clone** ( o : Object )

Clones any JSONable object.

**K3D gen**

Functions for generating basic 3D primitives. The output structure is as follows:

1{

2 verts : Array, // vertex coordinates

3 uvt : Array, // index coordinates

4 inds : Array // verts and uvt indices

5}

* **Plane** ( sx : int, sy : int) : Object

Generates a plane going from (-1,-1,0) to (1,1,0). **sx** - segments horizontally, **sy** - segments vertically.

* **Cube** ( ) : Object

Returns a 8-vertex cube going from (-1,-1,0) to (1,1,0).

* **Sphere** ( sx : int, sy : int ) : Object

Returns a centered sphere with radius 1. **sx** - segments horizontally, **sy** - segments vertically.

**K3D.edit**

* **interpolate** ( a : Array, b : Array, d : Array, t : Number)

Interpolates the values between a and b into d, using coefficient t.

* **transform** ( a : Array, m : Array)

Transforms the values from a into a using 4x4 matrix m.

* **unwrap** ( ind : Array, crd : Array, cpi : int ) : Array

Returns an array "crd", re-arranged by "ind", so it can be indexable with [0, 1, 2, 3, 4, ...]. "cpi" - coordinates per index. E.g.:

1{

2 // unwrapping square makes it have 6 vertices (3 triangles)

3 var a = K3D.edit.unwrap( [0,1,2,1,2,3], [0,0, 1,0, 0,1, 1,1], 2 );

4 // a = [0,0, 1,0, 0,1, 1,0, 0,1, 1,1]

5}

**K3D.mat**

Functions for generating 4x4 matrices.

* **scale** ( sx : Number, sy : Number, sz : Number) : Array

Generates scale matrix.

* **translate** ( tx : Number, ty : Number, tz : Number) : Array

Generates translation matrix.

* **rotate** ( rx : Number, ry : Number, rz : Number) : Array

Generates rotation matrix. Angles in radians.

* **rotateDeg** ( rx : Number, ry : Number, rz : Number) : Array

Generates rotation matrix. Angles in degrees.

**K3D.utils**

* **getAABB** ( vts : Array) : Object

Returns an axis-aligned bounding box for 3D vertices. The output is:

1{

2 min : { x : Number, y : Number, z : Number },

3 max : { x : Number, y : Number, z : Number }

4}

**K3D.parse**

All functions work with JSONable data. There is no universal format for 3D model, so each parser works with different structures.

* **fromJSON** ( buff : ArrayBuffer ) : Object

Parses any file from JSON format.

* **toJSON** ( o : Object ) : ArrayBuffer

Parses any file to JSON format.

* **fromOBJ** ( buff : ArrayBuffer ) : Object

Parses any file from OBJ format. The output structure is as follows:

1{

2 groups : { // groups in OBJ file

3 groupName : { from : int, to : int }, // index limits of group

4 ...

5 },

6

7 i\_verts : Array, // vertex indices

8 i\_uvt : Array, // texture indices

9 i\_norms : Array, // normal indices

10

11 c\_verts : Array, // vertex coordinates

12 c\_uvt : Array, // texture coordinates

13 c\_norms : Array, // normal coordinates

14}

* **fromMD2** ( buff : ArrayBuffer ) : Object

Parses any file from MD2 format. The output structure is as follows:

1{

2 frames : [ // frames in MD2 file

3 {

4 name : String, // frame name

5 norms : Array, // frame normals

6 verts : Array // frame vertices

7 },

8 ...

9 ],

10

11 skins : Array, // array of Strings

12

13 i\_verts : Array, // vertex indices

14 i\_uvt : Array, // texture indices

15

16 c\_uvt : Array, // texture coordinates

17}

* **from3DS** ( buff : ArrayBuffer ) : Object

Parses any file from 3DS format. The output structure is as follows:

1{

2 edit : {

3 objects : [

4 {

5 name : String, // object name

6 mesh : {

7 vertices : Array,

8 indices : Array,

9 uvt : Array

10 }

11 },

12 ...

13 ]

14 },

15

16 keyf : {

17 desc : [

18 {

19 hierarchy : {

20 name : String,

21 hierarchy : int

22 }

23 },

24 ...

25 ]

26 }

27}

* **fromCollada** ( buff : ArrayBuffer ) : Object

Parses any file from Collada format. Collada is very rich format and is not fully parsed. The output structure is as follows:

1{

2 asset : {

3 created : String,

4 modified : String,

5 up\_axis : String

6 },

7 geometries : [

8 {

9 sources : {

10 name1 : Array, // array of floats

11 name2 : Array, // array of floats

12 ...

13 },

14 triangles : [

15 {

16 i\_NORMAL : Array // normal indices

17 i\_TEXCOORD : Array // texture indices

18 i\_VERTEX : Array // vertex indices

19 s\_NORMAL : String // normal source

20 s\_TEXCOORD : String // texture source

21 s\_VERTEX : String // vertex source

22 material : String // material name

23 },

24 ...

25 ]

26 },

27 ...

28 ],

29 images : {

30 name1 : String,

31 ...

32 },

33 materials : {

34 name1 : String,

35 name2 : String,

36 ...

37 },

38 effects : {

39 name1 : {

40 surface : String

41 },

42 ...

43 }

44}