cglm Documentation

Release 0.8.3

Recep Aslantas

Jun 15, 2021

Getting Started:

cglm is optimized 3D math library written in C99 (compatible with C89). It is similar to original **glm** library except this is mainly for **C**

This library stores matrices as column-major order but in the future row-major is considered to be supported as optional.

Also currently only **float** type is supported for most operations.

CHAPTER 1

Features

- **scalar** and **simd** (sse, avx, neon…) optimizations
- option to use different clipspaces e.g. Left Handed, Zero-to-One… (currrently right handed negative-one is default)
- array api and struct api, you can use arrays or structs.
- general purpose matrix operations (mat4, mat3)
- chain matrix multiplication (square only)
- general purpose vector operations (cross, dot, rotate, proj, angle…)
- affine transformations
- matrix decomposition (extract rotation, scaling factor)
- optimized affine transform matrices (mul, rigid-body inverse)
- camera (lookat)
- projections (ortho, perspective)
- quaternions
- euler angles / yaw-pitch-roll to matrix
- extract euler angles
- inline or pre-compiled function call
- frustum (extract view frustum planes, corners…)
- bounding box (AABB in Frustum (culling), crop, merge…)
- bounding sphere
- project, unproject
- easing functions
- curves

- curve interpolation helpers (SMC, deCasteljau. . .)
- helpers to convert cglm types to Apple's simd library to pass cglm types to Metal GL without packing them on both sides
- ray intersection helpers
- and others. . .

CHAPTER 2

Build cglm

cglm does not have any external dependencies.

NOTE: If you only need to inline versions, you don't need to build **cglm**, you don't need to link it to your program. Just import cglm to your project as dependency / external lib by copy-paste then use it as usual

2.1 CMake (All platforms):

```
$ mkdir build
$ cd build
$ cmake .. # [Optional] -DCGLM_SHARED=ON
$ make
$ sudo make install # [Optional]
```

make will build cglm to **build** folder. If you don't want to install **cglm** to your system's folder you can get static and dynamic libs in this folder.

CMake Options:

```
option(CGLM_SHARED "Shared build" ON)
option(CGLM_STATIC "Static build" OFF)
option(CGLM_USE_C99 "" OFF) # C11
option(CGLM_USE_TEST "Enable Tests" OFF) # for make check - make test
```

Use as header-only library with your CMake project example This requires no building or installation of cglm.

```
cmake_minimum_required(VERSION 3.8.2)

project(<Your Project Name>)

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${LIBRARY_NAME} PRIVATE
```

(continues on next page)

(continued from previous page)

```
  cglm_headers)

add_subdirectory(external/cglm/ EXCLUDE_FROM_ALL)
```

Use with your CMake project example

```
cmake_minimum_required(VERSION 3.8.2)

project(<Your Project Name>)

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${LIBRARY_NAME} PRIVATE
  cglm)

add_subdirectory(external/cglm/)
```

2.2 Meson (All platforms):

Meson Options:

Use with your Meson project

2.3 Unix (Autotools):

```
$ sh autogen.sh
$ ./configure
$ make
$ make check            # run tests (optional)
$ [sudo] make install   # install to system (optional)
```

make will build cglm to **.libs** sub folder in project folder. If you don't want to install **cglm** to your system's folder you can get static and dynamic libs in this folder.

2.4 Windows (MSBuild):

Windows related build files, project files are located in *win* folder, make sure you are inside in cglm/win folder.

Code Analysis are enabled, it may take awhile to build.

```
$ cd win
$ .\build.bat
```

if *msbuild* is not worked (because of multi versions of Visual Studio) then try to build with *devenv*:

```
$ devenv cglm.sln /Build Release
```

Currently tests are not available on Windows.

2.5 Documentation (Sphinx):

cglm uses sphinx framework for documentation, it allows lot of formats for documentation. To see all options see sphinx build page:

https://www.sphinx-doc.org/en/master/man/sphinx-build.html

Example build:

```
$ cd cglm/docs
$ sphinx-build source build
```

CHAPTER 3

Getting Started

3.1 Types:

cglm uses **glm** prefix for all functions e.g. glm_lookat. You can see supported types in common header file:

```
typedef float                    vec2[2];
typedef float                    vec3[3];
typedef int                     ivec3[3];
typedef CGLM_ALIGN_IF(16) float vec4[4];
typedef vec4                     versor;
typedef vec3                     mat3[3];

#ifdef __AVX__
typedef CGLM_ALIGN_IF(32) vec4  mat4[4];
#else
typedef CGLM_ALIGN_IF(16) vec4  mat4[4];
#endif
```

As you can see types don't store extra informations in favor of space. You can send these values e.g. matrix to OpenGL directly without casting or calling a function like *value_ptr*

3.2 Alignment Is Required:

vec4 and **mat4** requires 16 (32 for **mat4** if AVX is enabled) byte alignment because **vec4** and **mat4** operations are vectorized by SIMD instructions (SSE/AVX/NEON).

UPDATE: By starting v0.4.5 cglm provides an option to disable alignment requirement, it is enabled as default

> Check *Options* page for more details

Also alignment is disabled for older msvc verisons as default. Now alignment is only required in Visual Studio 2017 version 15.6+ if CGLM_ALL_UNALIGNED macro is not defined.

3.3 Allocations:

cglm doesn't alloc any memory on heap. So it doesn't provide any allocator. You must allocate memory yourself. You should alloc memory for out parameters too if you pass pointer of memory location. When allocating memory, don't forget that **vec4** and **mat4** require alignment.

NOTE: Unaligned **vec4** and unaligned **mat4** operations will be supported in the future. Check todo list. Because you may want to multiply a CGLM matrix with external matrix. There is no guarantee that non-CGLM matrix is aligned. Unaligned types will have *u* prefix e.g. **umat4**

3.4 Array vs Struct:

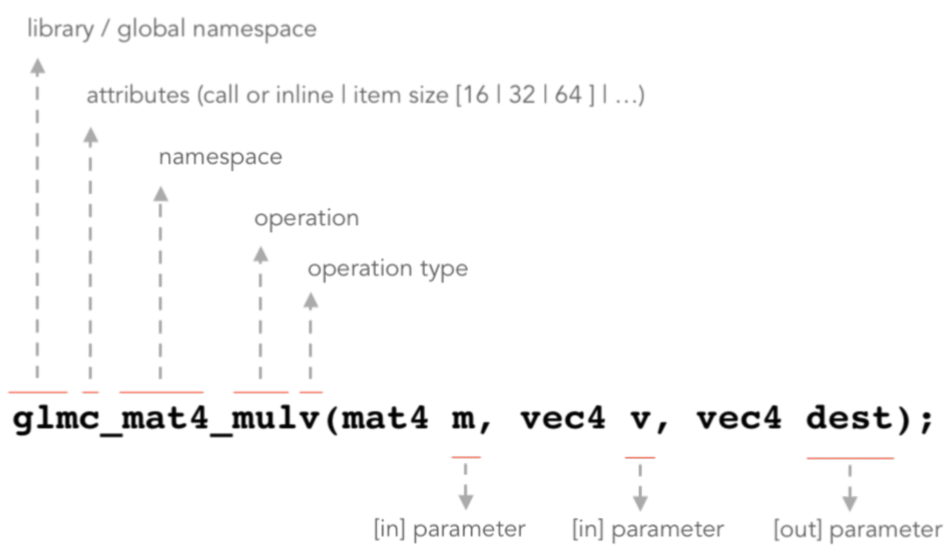
cglm uses arrays for vector and matrix types. So you can't access individual elements like vec.x, vec.y, vec.z... You must use subscript to access vector elements e.g. vec[0], vec[1], vec[2].

Also I think it is more meaningful to access matrix elements with subscript e.g **matrix[2][3]** instead of **matrix._23**. Since matrix is array of vectors, vectors are also defined as array. This makes types homogeneous.

Return arrays?

Since C doesn't support return arrays, cglm also doesn't support this feature.

3.5 Function design:



cglm provides a few way to call a function to do same operation.

- Inline - *glm_*, *glm_u*
- Pre-compiled - *glmc_*, *glmc_u*

For instance **glm_mat4_mul** is inline (all *glm_* functions are inline), to make it non-inline (pre-compiled), call it as **glmc_mat4_mul** from library, to use unaligned version use **glm_umat4_mul** (todo).

Most functions have **dest** parameter for output. For instance mat4_mul func looks like this:

```
CGLM_INLINE
void
glm_mat4_mul(mat4 m1, mat4 m2, mat4 dest)
```

The dest parameter is out parameter. Result will be stored in **dest**. Also in this case matrix multiplication order is dest = m1 * m2.

- Changing parameter order will change the multiplication order.
- You can pass all parameter same (this is similar to m1 *= m1), you can pass **dest** as m1 or m2 (this is similar to m1 *= m2)

3.5.1 v postfix in function names

You may see **v** postfix in some function names, v stands for vector. For instance consider a function that accepts three parameters x, y, z. This function may be overloaded by **v** postfix to accept vector (vec3) instead of separate parameters. In some places the v means that it will be apply to a vector.

3.5.2 _to postfix in function names

_to version of function will store the result in specified parameter instead of in-out parameter. Some functions don't have _to prefix but they still behave like this e.g. glm_mat4_mul.

CHAPTER 4

How to send vector or matrix to OpenGL like API

cglm's vector and matrix types are arrays. So you can send them directly to a function which accepts pointer. But you may got warnings for matrix because it is two dimensional array.

4.1 Passing / Uniforming Matrix to OpenGL:

glUniformMatrix4fv accepts float pointer, you can pass matrix to that parameter and it should work but with warnings. "You can pass" doesn't mean that you must pass like that.

Correct options:

Correct doesn't mean correct way to use OpenGL it is just shows correct way to pass cglm type to it.

4.1.1 1. Pass first column

The goal is that pass address of matrix, first element of matrix is also address of matrix, because it is array of vectors and vector is array of floats.

```
mat4 matrix;
/* ... */
glUniformMatrix4fv(location, 1, GL_FALSE, matrix[0]);
```

array of matrices:

```
mat4 matrix;
/* ... */
glUniformMatrix4fv(location, count, GL_FALSE, matrix[0][0]);
```

4.1.2 1. Cast matrix to pointer

```
mat4 matrix;
/* ... */
glUniformMatrix4fv(location, count, GL_FALSE, (float *)matrix);
```

in this way, passing aray of matrices is same

4.2 Passing / Uniforming Vectors to OpenGL:

You don't need to do extra thing when passing cglm vectors to OpengL or other APIs. Because a function like **glUniform4fv** accepts vector as pointer. cglm's vectors are array of floats. So you can pass it directly ot those functions:

```
vec4 vec;
/* ... */
glUniform4fv(location, 1, vec);
```

this show how to pass **vec4** others are same.

CHAPTER 5

API documentation

Some functions may exist twice, once for their namespace and once for global namespace to make easier to write very common functions

For instance, in general we use `glm_vec3_dot` to get dot product of two **vec3**. Now we can also do this with `glm_dot`, same for *_cross* and so on. . .

The original function stays where it is, the function in global namespace of same name is just an alias, so there is no call version of those functions. e.g there is no func like `glmc_dot` because *glm_dot* is just alias for `glm_vec3_dot`

By including **cglm/cglm.h** header you will include all inline version of functions. Since functions in this header[s] are inline you don't need to build or link *cglm* against your project.

But by including **cglm/call.h** header you will include all *non-inline* version of functions. You need to build *cglm* and link it. Follow the *Build cglm* documentation for this

5.1 3D Affine Transforms

Header: cglm/affine.h

5.1.1 Initialize Transform Matrices

Functions with **_make** prefix expect you don't have a matrix and they create a matrix for you. You don't need to pass identity matrix.

But other functions expect you have a matrix and you want to transform them. If you didn't have any existing matrix you have to initialize matrix to identity before sending to transfrom functions.

There are also functions to decompose transform matrix. These functions can't decompose matrix after projected.

5.1.2 Rotation Center

Rotating functions uses origin as rotation center (pivot/anchor point), since scale factors are stored in rotation matrix, same may also true for scalling. cglm provides some functions for rotating around at given point e.g. **glm_rotate_at**, **glm_quat_rotate_at**. Use them or follow next section for algorihm ("Rotate or Scale around specific Point (Pivot Point / Anchor Point)").

5.1.3 Rotate or Scale around specific Point (Anchor Point)

If you want to rotate model around arbibtrary point follow these steps:

1. Move model from pivot point to origin: **translate(-pivot.x, -pivot.y, -pivot.z)**
2. Apply rotation (or scaling maybe)
3. Move model back from origin to pivot (reverse of step-1): **translate(pivot.x, pivot.y, pivot.z)**

glm_rotate_at, **glm_quat_rotate_at** and their helper functions works that way.

The implementation would be:

```
glm_translate(m, pivot);
glm_rotate(m, angle, axis);
glm_translate(m, pivotInv); /* pivotInv = -pivot */
```

5.1.4 Transforms Order

It is important to understand this part especially if you call transform functions multiple times

glm_translate, *glm_rotate*, *glm_scale* and *glm_quat_rotate* and their helpers functions works like this (cglm may provide reverse order too as alternative in the future):

```
TransformMatrix = TransformMatrix * TraslateMatrix; // glm_translate()
TransformMatrix = TransformMatrix * RotateMatrix;   // glm_rotate(), glm_quat_rotate()
TransformMatrix = TransformMatrix * ScaleMatrix;    // glm_scale()
```

As you can see it is multipled as right matrix. For instance what will happen if you call *glm_translate* twice?

```
glm_translate(transform, translate1); /* transform = transform * translate1 */
glm_translate(transform, translate2); /* transform = transform * translate2 */
glm_rotate(transform, angle, axis)    /* transform = transform * rotation   */
```

Now lets try to understand this:

1. You call translate using *translate1* and you expect it will be first transform because you call it first, do you?

Result will be **'transform = transform * translate1'**

2. Then you call translate using *translate2* and you expect it will be second transform?

Result will be **'transform = transform * translate2'**. Now lets expand transform, it was *transform* * *translate1* before second call.

Now it is **'transform = transform * translate1 * translate2'**, now do you understand what I say?

3. After last call transform will be:

‘transform = transform * translate1 * translate2 * rotation‘

The order will be; **rotation will be applied first**, then **translate2** then **translate1**

It is all about matrix multiplication order. It is similar to MVP matrix: *MVP = Projection * View * Model*, model will be applied first, then view then projection.

Confused?

In the end the last function call applied first in shaders.

As alternative way, you can create transform matrices individually then combine manually, but don’t forget that *glm_translate*, *glm_rotate*, *glm_scale*… are optimized and should be faster (an smaller assembly output) than manual multiplication

```
mat4 transform1, transform2, transform3, finalTransform;

glm_translate_make(transform1, translate1);
glm_translate_make(transform2, translate2);
glm_rotate_make(transform3, angle, axis);

/* first apply transform1, then transform2, thentransform3 */
glm_mat4_mulN((mat4 *[]){&transform3, &transform2, &transform1}, 3, finalTransform);

/* if you don't want to use mulN, same as above */
glm_mat4_mul(transform3, transform2, finalTransform);
glm_mat4_mul(finalTransform, transform1, finalTransform);
```

Now transform1 will be applied first, then transform2 then transform3

5.1.5 Table of contents (click to go):

Functions:

17. *glm_rotate_atm()*
18. *glm_decompose_scalev()*
19. *glm_uniscaled()*
20. *glm_decompose_rs()*
21. *glm_decompose()*

5.1.6 Functions documentation

void **glm_translate_to**(mat4 *m*, vec3 *v*, mat4 *dest*)
translate existing transform matrix by *v* vector and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **v** translate vector [x, y, z]

[out] **dest** translated matrix

void **glm_translate**(mat4 *m*, vec3 *v*)
translate existing transform matrix by *v* vector and stores result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **v** translate vector [x, y, z]

void **glm_translate_x**(mat4 *m*, float *x*)
translate existing transform matrix by x factor

Parameters:

[in, out] **m** affine transfrom

[in] **v** x factor

void **glm_translate_y**(mat4 *m*, float *y*)
translate existing transform matrix by *y* factor

Parameters:

[in, out] **m** affine transfrom

[in] **v** y factor

void **glm_translate_z**(mat4 *m*, float *z*)
translate existing transform matrix by *z* factor

Parameters:

[in, out] **m** affine transfrom

[in] **v** z factor

void **glm_translate_make**(mat4 *m*, vec3 *v*)
creates NEW translate transform matrix by *v* vector.

Parameters:

[in, out] **m** affine transfrom

[in] **v** translate vector [x, y, z]

void **glm_scale_to**(mat4 *m*, vec3 *v*, mat4 *dest*)
scale existing transform matrix by *v* vector and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **v** scale vector [x, y, z]

[out] **dest** scaled matrix

void **glm_scale_make**(mat4 *m*, vec3 *v*)

creates NEW scale matrix by v vector

Parameters:

[out] **m** affine transfrom

[in] **v** scale vector [x, y, z]

void **glm_scale**(mat4 *m*, vec3 *v*)

scales existing transform matrix by v vector and stores result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **v** scale vector [x, y, z]

void **glm_scale_uni**(mat4 *m*, float *s*)

applies uniform scale to existing transform matrix v = [s, s, s] and stores result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **v** scale factor

void **glm_rotate_x**(mat4 *m*, float *angle*, mat4 *dest*)

rotate existing transform matrix around X axis by angle and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **angle** angle (radians)

[out] **dest** rotated matrix

void **glm_rotate_y**(mat4 *m*, float *angle*, mat4 *dest*)

rotate existing transform matrix around Y axis by angle and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **angle** angle (radians)

[out] **dest** rotated matrix

void **glm_rotate_z**(mat4 *m*, float *angle*, mat4 *dest*)

rotate existing transform matrix around Z axis by angle and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **angle** angle (radians)

[out] **dest** rotated matrix

void **glm_rotate_make**(mat4 *m*, float *angle*, vec3 *axis*)

creates NEW rotation matrix by angle and axis, axis will be normalized so you don't need to normalize it

Parameters:

[out] **m** affine transfrom

[in] **axis** angle (radians)

[in] **axis** axis

void **glm_rotate** (mat4 *m*, float *angle*, vec3 *axis*)

rotate existing transform matrix around Z axis by angle and axis

Parameters:

[in, out] **m** affine transfrom

[in] **angle** angle (radians)

[in] **axis** axis

void **glm_rotate_at** (mat4 *m*, vec3 *pivot*, float *angle*, vec3 *axis*)

rotate existing transform around given axis by angle at given pivot point (rotation center)

Parameters:

[in, out] **m** affine transfrom

[in] **pivot** pivot, anchor point, rotation center

[in] **angle** angle (radians)

[in] **axis** axis

void **glm_rotate_atm** (mat4 *m*, vec3 *pivot*, float *angle*, vec3 *axis*)

creates NEW rotation matrix by angle and axis at given point

this creates rotation matrix, it assumes you don't have a matrix

this should work faster than glm_rotate_at because it reduces one glm_translate.

Parameters:

[in, out] **m** affine transfrom

[in] **pivot** pivot, anchor point, rotation center

[in] **angle** angle (radians)

[in] **axis** axis

void **glm_decompose_scalev** (mat4 *m*, vec3 *s*)

decompose scale vector

Parameters:

[in] **m** affine transform

[out] **s** scale vector (Sx, Sy, Sz)

bool **glm_uniscaled** (mat4 *m*)

returns true if matrix is uniform scaled. This is helpful for creating normal matrix.

Parameters:

[in] **m** matrix

void **glm_decompose_rs** (mat4 *m*, mat4 *r*, vec3 *s*)

decompose rotation matrix (mat4) and scale vector [Sx, Sy, Sz] DON'T pass projected matrix here

Parameters:

[in] **m** affine transform

[out] **r** rotation matrix

[out] **s** scale matrix

void **glm_decompose**(mat4 *m*, vec4 *t*, mat4 *r*, vec3 *s*)

decompose affine transform, TODO: extract shear factors. DON'T pass projected matrix here

Parameters:

[in] **m** affine transfrom

[out] **t** translation vector

[out] **r** rotation matrix (mat4)

[out] **s** scaling vector [X, Y, Z]

5.2 3D Affine Transform Matrix (specialized functions)

Header: cglm/affine-mat.h

We mostly use glm_mat4_* for 4x4 general and transform matrices. **cglm** provides optimized version of some functions. Because affine transform matrix is a known format, for instance all last item of first three columns is zero.

You should be careful when using these functions. For instance `glm_mul()` assumes matrix will be this format:

```
R  R  R  X
R  R  R  Y
R  R  R  Z
0  0  0  W
```

if you override zero values here then use `glm_mat4_mul()` version. You cannot use `glm_mul()` anymore.

Same is also true for `glm_inv_tr()` if you only have rotation and translation then it will work as expected, otherwise you cannot use that.

In the future it may accept scale factors too but currectly it does not.

5.2.1 Table of contents (click func go):

Functions:

5.2.2 Functions documentation

void **glm_mul**(mat4 *m1*, mat4 *m2*, mat4 *dest*)

this is similar to glm_mat4_mul but specialized to affine transform

Matrix format should be:

```
R  R  R  X
R  R  R  Y
R  R  R  Z
0  0  0  W
```

this reduces some multiplications. It should be faster than mat4_mul. if you are not sure about matrix format then DON'T use this! use mat4_mul

Parameters:

[in] **m1** affine matrix 1

[in] **m2** affine matrix 2

[out] **dest** result matrix

void **glm_mul_rot** (mat4 *m1*, mat4 *m2*, mat4 *dest*)

this is similar to glm_mat4_mul but specialized to rotation matrix

Right Matrix format should be (left is free):

```
R  R  R  0
R  R  R  0
R  R  R  0
0  0  0  1
```

this reduces some multiplications. It should be faster than mat4_mul. if you are not sure about matrix format then DON'T use this! use mat4_mul

Parameters:

[in] **m1** affine matrix 1

[in] **m2** affine matrix 2

[out] **dest** result matrix

void **glm_inv_tr** (mat4 *mat*)

inverse orthonormal rotation + translation matrix (ridig-body)

```
X = | R  T |   X' = | R' -R'T |
    | 0  1 |        | 0     1 |
```

use this if you only have rotation + translation, this should work faster than `glm_mat4_inv()`

Don't use this if your matrix includes other things e.g. scale, shear. . .

Parameters:

[in,out] **mat** affine matrix

5.3 2D Affine Transforms

Header: cglm/affine2d.h

2D Transforms uses *2d* suffix for naming. If there is no 2D suffix it is 3D function.

5.3.1 Initialize Transform Matrices

Functions with **_make** prefix expect you don't have a matrix and they create a matrix for you. You don't need to pass identity matrix.

But other functions expect you have a matrix and you want to transform them. If you didn't have any existing matrix you have to initialize matrix to identity before sending to transfrom functions.

5.3.2 Transforms Order

See *Transforms Order* to read similar section.

5.3.3 Table of contents (click to go):

Functions:

void **glm_translate2d**(mat3 *m*, vec2 *v*)

translate existing 2d transform matrix by *v* vector and stores result in same matrix

Parameters:

- *[in, out]* **m** 2d affine transfrom
- *[in]* **v** translate vector [x, y]

void **glm_translate2d_to**(mat3 *m*, vec2 *v*, mat3 *dest*)

translate existing 2d transform matrix by *v* vector and store result in dest

Parameters:

- *[in]* **m** 2d affine transfrom
- *[in]* **v** translate vector [x, y]
- *[out]* **dest** translated matrix

void **glm_translate2d_x**(mat3 *m*, float *x*)

translate existing 2d transform matrix by x factor

Parameters:

- *[in, out]* **m** 2d affine transfrom
- *[in]* **x** x factor

void **glm_translate2d_y**(mat3 *m*, float *y*)

translate existing 2d transform matrix by y factor

Parameters:

- *[in, out]* **m** 2d affine transfrom
- *[in]* **y** y factor

void **glm_translate2d_make** (mat3 *m*, vec2 *v*)
creates NEW translate 2d transform matrix by *v* vector

Parameters:

[in, out] **m** affine transfrom

[in] **v** translate vector [x, y]

void **glm_scale2d_to** (mat3 *m*, vec2 *v*, mat3 *dest*)
scale existing 2d transform matrix by *v* vector and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **v** scale vector [x, y]

[out] **dest** scaled matrix

void **glm_scale2d_make** (mat3 *m*, vec2 *v*)
creates NEW 2d scale matrix by *v* vector

Parameters:

[in, out] **m** affine transfrom

[in] **v** scale vector [x, y]

void **glm_scale2d** (mat3 *m*, vec2 *v*)
scales existing 2d transform matrix by *v* vector and stores result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **v** translate vector [x, y]

void **glm_scale2d_uni** (mat3 *m*, float *s*)
applies uniform scale to existing 2d transform matrix v = [s, s] and stores result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **s** scale factor

void **glm_rotate2d_make** (mat3 *m*, float *angle*)
creates NEW rotation matrix by angle around *Z* axis

Parameters:

[in, out] **m** affine transfrom

[in] **angle** angle (radians)

void **glm_rotate2d** (mat3 *m*, float *angle*)
rotate existing 2d transform matrix around *Z* axis by angle and store result in same matrix

Parameters:

[in, out] **m** affine transfrom

[in] **angle** angle (radians)

void **glm_rotate2d_to** (mat3 *m*, float *angle*, mat3 *dest*)
rotate existing 2d transform matrix around *Z* axis by angle and store result in dest

Parameters:

[in] **m** affine transfrom

[in] **angle** angle (radians)

[out] **dest** rotated matrix

5.4 camera

Header: cglm/cam.h

There are many convenient functions for camera. For instance `glm_look()` is just wrapper for `glm_lookat()`. Sometimes you only have direction instead of target, so that makes easy to build view matrix using direction. There is also `glm_look_anyup()` function which can help build view matrix without providing UP axis. It uses `glm_vec3_ortho()` to get a UP axis and builds view matrix.

You can also *_default* versions of ortho and perspective to build projection fast if you don't care specific projection values.

_decomp means decompose; these function can help to decompose projection matrices.

> **NOTE**: Be careful when working with high range (very small near, very large far) projection matrices. You may not get exact value you gave. **float** type cannot store very high precision so you will lose precision. Also your projection matrix will be inaccurate due to losing precision

5.4.1 Table of contents (click to go):

Functions:

21. *glm_persp_decomp_near()*
22. *glm_persp_fovy()*
23. *glm_persp_aspect()*
24. *glm_persp_sizes()*

5.4.2 Functions documentation

void **glm_frustum**(float *left*, float *right*, float *bottom*, float *top*, float *nearVal*, float *farVal*, mat4 *dest*)

set up perspective peprojection matrix

Parameters:

[in] **left** viewport.left
[in] **right** viewport.right
[in] **bottom** viewport.bottom
[in] **top** viewport.top
[in] **nearVal** near clipping plane
[in] **farVal** far clipping plane
[out] **dest** result matrix

void **glm_ortho**(float *left*, float *right*, float *bottom*, float *top*, float *nearVal*, float *farVal*, mat4 *dest*)

set up orthographic projection matrix

Parameters:

[in] **left** viewport.left
[in] **right** viewport.right
[in] **bottom** viewport.bottom
[in] **top** viewport.top
[in] **nearVal** near clipping plane
[in] **farVal** far clipping plane
[out] **dest** result matrix

void **glm_ortho_aabb**(vec3 *box[2]*, mat4 *dest*)

set up orthographic projection matrix using bounding box
bounding box (AABB) must be in view space

Parameters:

[in] **box** AABB
[in] **dest** result matrix

void **glm_ortho_aabb_p**(vec3 *box[2]*, float *padding*, mat4 *dest*)

set up orthographic projection matrix using bounding box
bounding box (AABB) must be in view space

this version adds padding to box

Parameters:

[in] **box** AABB
[in] **padding** padding
[out] **d** result matrix

void **glm_ortho_aabb_pz** (vec3 *box[2]*, float *padding*, mat4 *dest*)

set up orthographic projection matrix using bounding box
bounding box (AABB) must be in view space

this version adds Z padding to box

Parameters:

[in] **box** AABB
[in] **padding** padding for near and far
[out] **d** result matrix

Returns: square of norm / magnitude

void **glm_ortho_default** (float *aspect*, mat4 *dest*)

set up unit orthographic projection matrix

Parameters:

[in] **aspect** aspect ration (width / height)
[out] **dest** result matrix

void **glm_ortho_default_s** (float *aspect*, float *size*, mat4 *dest*)

set up orthographic projection matrix with given CUBE size

Parameters:

[in] **aspect** aspect ration (width / height)
[in] **size** cube size
[out] **dest** result matrix

void **glm_perspective** (float *fovy*, float *aspect*, float *nearVal*, float *farVal*, mat4 *dest*)

set up perspective projection matrix

Parameters:

[in] **fovy** field of view angle
[in] **aspect** aspect ratio (width / height)
[in] **nearVal** near clipping plane
[in] **farVal** far clipping planes
[out] **dest** result matrix

void **glm_persp_move_far** (mat4 *proj*, float *deltaFar*) extend perspective projection matrix's far distance

this function does not guarantee far >= near, be aware of that!

Parameters:

[in, out] **proj** projection matrix to extend

[in] **deltaFar** distance from existing far (negative to shink)

void **glm_perspective_default** (float *aspect*, mat4 *dest*)

set up perspective projection matrix with default near/far and angle values

Parameters:

[in] **aspect** aspect aspect ratio (width / height)

[out] **dest** result matrix

void **glm_perspective_resize** (float *aspect*, mat4 *proj*)

resize perspective matrix by aspect ratio (width / height) this makes very easy to resize proj matrix when window / viewport reized

Parameters:

[in] **aspect** aspect ratio (width / height)

[in, out] **proj** perspective projection matrix

void **glm_lookat** (vec3 *eye*, vec3 *center*, vec3 *up*, mat4 *dest*)

set up view matrix

NOTE: The UP vector must not be parallel to the line of sight from the eye point to the reference point.

Parameters:

[in] **eye** eye vector

[in] **center** center vector

[in] **up** up vector

[out] **dest** result matrix

void **glm_look** (vec3 *eye*, vec3 *dir*, vec3 *up*, mat4 *dest*)

set up view matrix

convenient wrapper for `glm_lookat()`: if you only have direction not target self then this might be useful. Because you need to get target from direction.

NOTE: The UP vector must not be parallel to the line of sight from the eye point to the reference point.

Parameters:

[in] **eye** eye vector

[in] **dir** direction vector

[in] **up** up vector

[out] **dest** result matrix

void **glm_look_anyup**(vec3 *eye*, vec3 *dir*, mat4 *dest*)

set up view matrix

convenient wrapper for `glm_look()` if you only have direction and if you don't care what UP vector is then this might be useful to create view matrix

Parameters:

[in] **eye** eye vector

[in] **dir** direction vector

[out] **dest** result matrix

void **glm_persp_decomp**(mat4 *proj*, float **nearVal*, float **farVal*, float **top*, float **bottom*, float **left*, float **right*)

decomposes frustum values of perspective projection.

Parameters:

[in] **eye** perspective projection matrix

[out] **nearVal** near

[out] **farVal** far

[out] **top** top

[out] **bottom** bottom

[out] **left** left

[out] **right** right

void **glm_persp_decompv**(mat4 *proj*, float *dest[6]*)

decomposes frustum values of perspective projection.
this makes easy to get all values at once

Parameters:

[in] **proj** perspective projection matrix

[out] **dest** array

void **glm_persp_decomp_x**(mat4 *proj*, float **left*, float **right*)

decomposes left and right values of perspective projection.
x stands for x axis (left / right axis)

Parameters:

[in] **proj** perspective projection matrix

[out] **left** left

[out] **right** right

void **glm_persp_decomp_y**(mat4 *proj*, float **top*, float **bottom*)

decomposes top and bottom values of perspective projection.
y stands for y axis (top / botom axis)

Parameters:

[in] **proj** perspective projection matrix
[out] **top** top
[out] **bottom** bottom

void **glm_persp_decomp_z** (mat4 *proj*, float **nearVal*, float **farVal*)

decomposes near and far values of perspective projection.
z stands for z axis (near / far axis)

Parameters:

[in] **proj** perspective projection matrix
[out] **nearVal** near
[out] **farVal** far

void **glm_persp_decomp_far** (mat4 *proj*, float * __restrict *farVal*)

decomposes far value of perspective projection.

Parameters:

[in] **proj** perspective projection matrix
[out] **farVal** far

void **glm_persp_decomp_near** (mat4 *proj*, float * __restrict *nearVal*)

decomposes near value of perspective projection.

Parameters:

[in] **proj** perspective projection matrix
[out] **nearVal** near

float **glm_persp_fovy** (mat4 *proj*)

returns field of view angle along the Y-axis (in radians)

if you need to degrees, use glm_deg to convert it or use this: fovy_deg = glm_deg(glm_persp_fovy(projMatrix))

Parameters:

[in] **proj** perspective projection matrix

Returns:

fovy in radians

float **glm_persp_aspect** (mat4 *proj*)

returns aspect ratio of perspective projection

Parameters:

[in] **proj** perspective projection matrix

void **glm_persp_sizes** (mat4 *proj*, float *fovy*, vec4 *dest*)

returns sizes of near and far planes of perspective projection

Parameters:

[in] **proj** perspective projection matrix
[in] **fovy** fovy (see brief)
[out] **dest** sizes order: [Wnear, Hnear, Wfar, Hfar]

5.5 frustum

Header: cglm/frustum.h

cglm provides convenient functions to extract frustum planes, corners... All extracted corners are **vec4** so you must create array of **vec4** not **vec3**. If you want to store them to save space you msut convert them yourself.

vec4 is used to speed up functions need to corners. This is why frustum fucntions use *vec4* instead of *vec3*

Currenty related-functions use [-1, 1] clip space configuration to extract corners but you can override it by prodiving **GLM_CUSTOM_CLIPSPACE** macro. If you provide it then you have to all bottom macros as *vec4*

Current configuration:

```
/* near */
GLM_CSCOORD_LBN {-1.0f, -1.0f, -1.0f, 1.0f}
GLM_CSCOORD_LTN {-1.0f,  1.0f, -1.0f, 1.0f}
GLM_CSCOORD_RTN { 1.0f,  1.0f, -1.0f, 1.0f}
GLM_CSCOORD_RBN { 1.0f, -1.0f, -1.0f, 1.0f}

/* far */
GLM_CSCOORD_LBF {-1.0f, -1.0f,  1.0f, 1.0f}
GLM_CSCOORD_LTF {-1.0f,  1.0f,  1.0f, 1.0f}
GLM_CSCOORD_RTF { 1.0f,  1.0f,  1.0f, 1.0f}
GLM_CSCOORD_RBF { 1.0f, -1.0f,  1.0f, 1.0f}
```

Explain of short names:

- **LBN**: left bottom near
- **LTN**: left top near
- **RTN**: right top near
- **RBN**: right bottom near
- **LBF**: left bottom far
- **LTF**: left top far
- **RTF**: right top far
- **RBF**: right bottom far

5.5.1 Table of contents (click to go):

Macros:

```
GLM_LBN     0 /* left  bottom near */
GLM_LTN     1 /* left  top    near */
GLM_RTN     2 /* right top    near */
GLM_RBN     3 /* right bottom near */

GLM_LBF     4 /* left  bottom far  */
GLM_LTF     5 /* left  top    far  */
GLM_RTF     6 /* right top    far  */
GLM_RBF     7 /* right bottom far  */

GLM_LEFT    0
GLM_RIGHT   1
GLM_BOTTOM  2
GLM_TOP     3
GLM_NEAR    4
GLM_FAR     5
```

Functions:

1. *glm_frustum_planes()*
2. *glm_frustum_corners()*
3. *glm_frustum_center()*
4. *glm_frustum_box()*
5. *glm_frustum_corners_at()*

5.5.2 Functions documentation

void **glm_frustum_planes**(mat4 *m*, vec4 *dest[6]*)

extracts view frustum planes

planes' space:

- if m = proj: View Space
- if m = viewProj: World Space
- if m = MVP: Object Space

You probably want to extract planes in world space so use viewProj as m Computing viewProj:

```
glm_mat4_mul(proj, view, viewProj);
```

Exracted planes order: [left, right, bottom, top, near, far]

Parameters:

[in] **m** matrix
[out] **dest** exracted view frustum planes

void **glm_frustum_corners**(mat4 *invMat*, vec4 *dest[8]*)

extracts view frustum corners using clip-space coordinates

corners' space:

- if m = invViewProj: World Space
- if m = invMVP: Object Space

You probably want to extract corners in world space so use **invViewProj** Computing invViewProj:

```
glm_mat4_mul(proj, view, viewProj);
...
glm_mat4_inv(viewProj, invViewProj);
```

if you have a near coord at **i** index, you can get it's far coord by i + 4; follow example below to understand that

For instance to find center coordinates between a near and its far coord:

```
for (j = 0; j < 4; j++) {
  glm_vec3_center(corners[i], corners[i + 4], centerCorners[i]);
}
```

corners[i + 4] is far of corners[i] point.

Parameters:

[in] **invMat** matrix

[out] **dest** exracted view frustum corners

void **glm_frustum_center**(vec4 *corners[8]*, vec4 *dest*)

finds center of view frustum

Parameters:

[in] **corners** view frustum corners

[out] **dest** view frustum center

void **glm_frustum_box**(vec4 *corners[8]*, mat4 *m*, vec3 *box[2]*)

finds bounding box of frustum relative to given matrix e.g. view mat

Parameters:

[in] **corners** view frustum corners

[in] **m** matrix to convert existing conners

[out] **box** bounding box as array [min, max]

void **glm_frustum_corners_at**(vec4 *corners[8]*, float *splitDist*, float *farDist*, vec4 *planeCorners[4]*)

finds planes corners which is between near and far planes (parallel)

this will be helpful if you want to split a frustum e.g. CSM/PSSM. This will find planes' corners but you will need to one more plane. Actually you have it, it is near, far or created previously with this func ;)

Parameters:

[in] **corners** frustum corners

[in] **splitDist** split distance

[in] **farDist** far distance (zFar)

[out] **planeCorners** plane corners [LB, LT, RT, RB]

5.6 axis aligned bounding box (AABB)

Header: cglm/box.h

Some convenient functions provided for AABB.

Definition of box:

cglm defines box as two dimensional array of vec3. The first element is **min** point and the second one is **max** point. If you have another type e.g. struct or even another representation then you must convert it before and after call cglm box function.

5.6.1 Table of contents (click to go):

Functions:

1. `glm_aabb_transform()`
2. `glm_aabb_merge()`
3. `glm_aabb_crop()`
4. `glm_aabb_crop_until()`
5. `glm_aabb_frustum()`
6. `glm_aabb_invalidate()`
7. `glm_aabb_isvalid()`
8. `glm_aabb_size()`
9. `glm_aabb_radius()`
10. `glm_aabb_center()`
11. `glm_aabb_aabb()`
12. `glm_aabb_sphere()`
13. `glm_aabb_point()`
14. `glm_aabb_contains()`

5.6.2 Functions documentation

void **glm_aabb_transform**(vec3 *box[2]*, mat4 *m*, vec3 *dest[2]*)

apply transform to Axis-Aligned Bounding Box

Parameters:

[in] **box** bounding box
[in] **m** transform matrix
[out] **dest** transformed bounding box

void **glm_aabb_merge**(vec3 *box1[2]*, vec3 *box2[2]*, vec3 *dest[2]*)

merges two AABB bounding box and creates new one

two box must be in same space, if one of box is in different space then you should consider to convert it's space by glm_box_space

Parameters:

[in] **box1** bounding box 1

[in] **box2** bounding box 2

[out] **dest** merged bounding box

void **glm_aabb_crop**(vec3 *box[2]*, vec3 *cropBox[2]*, vec3 *dest[2]*)

crops a bounding box with another one.

this could be useful for gettng a bbox which fits with view frustum and object bounding boxes. In this case you crop view frustum box with objects box

Parameters:

[in] **box** bounding box 1

[in] **cropBox** crop box

[out] **dest** cropped bounding box

void **glm_aabb_crop_until**(vec3 *box[2]*, vec3 *cropBox[2]*, vec3 *clampBox[2]*, vec3 *dest[2]*)

crops a bounding box with another one.

this could be useful for gettng a bbox which fits with view frustum and object bounding boxes. In this case you crop view frustum box with objects box

Parameters:

[in] **box** bounding box

[in] **cropBox** crop box

[in] **clampBox** miniumum box

[out] **dest** cropped bounding box

bool **glm_aabb_frustum**(vec3 *box[2]*, vec4 *planes[6]*)

check if AABB intersects with frustum planes

this could be useful for frustum culling using AABB.

OPTIMIZATION HINT: if planes order is similar to LEFT, RIGHT, BOTTOM, TOP, NEAR, FAR then this method should run even faster because it would only use two planes if object is not inside the two planes fortunately cglm extracts planes as this order! just pass what you got!

Parameters:

[in] **box** bounding box

[out] **planes** frustum planes

void **glm_aabb_invalidate**(vec3 *box[2]*)

invalidate AABB min and max values

It fills *max* values with -FLT_MAX and *min* values with +FLT_MAX

Parameters:

[in, out] **box** bounding box

bool **glm_aabb_isvalid**(vec3 *box[2]*)

check if AABB is valid or not

Parameters:

[in] **box** bounding box

Returns: returns true if aabb is valid otherwise false

float **glm_aabb_size**(vec3 *box[2]*)

distance between of min and max

Parameters:

[in] **box** bounding box

Returns: distance between min - max

float **glm_aabb_radius**(vec3 *box[2]*)

radius of sphere which surrounds AABB

Parameters:

[in] **box** bounding box

void **glm_aabb_center**(vec3 *box[2]*, vec3 *dest*)

computes center point of AABB

Parameters:

[in] **box** bounding box
[out] **dest** center of bounding box

bool **glm_aabb_aabb**(vec3 *box[2]*, vec3 *other[2]*)

check if two AABB intersects

Parameters:

[in] **box** bounding box
[out] **other** other bounding box

bool **glm_aabb_sphere**(vec3 *box[2]*, vec4 *s*) check if AABB intersects with sphere

https://github.com/erich666/GraphicsGems/blob/master/gems/BoxSphere.c
Solid Box - Solid Sphere test.

Parameters:

[in] **box** solid bounding box
[out] **s** solid sphere

bool **`glm_aabb_point`** (vec3 *box[2]*, vec3 *point*)

check if point is inside of AABB

Parameters:

[in] **box** bounding box
[out] **point** point

bool **`glm_aabb_contains`** (vec3 *box[2]*, vec3 *other[2]*)

check if AABB contains other AABB

Parameters:

[in] **box** bounding box
[out] **other** other bounding box

5.7 quaternions

Header: cglm/quat.h

Important: *cglm* stores quaternion as **[x, y, z, w]** in memory since **v0.4.0** it was **[w, x, y, z]** before v0.4.0 (**v0.3.5 and earlier**). w is real part.

What you can do with quaternions with existing functions is (Some of them):

- You can rotate transform matrix using quaterion
- You can rotate vector using quaterion
- You can create view matrix using quaterion
- You can create a lookrotation (from source point to dest)

5.7.1 Table of contents (click to go):

Macros:

1. GLM_QUAT_IDENTITY_INIT
2. GLM_QUAT_IDENTITY

Functions:

1. *glm_quat_identity()*
2. *glm_quat_identity_array()*
3. *glm_quat_init()*
4. *glm_quat()*
5. *glm_quatv()*
6. *glm_quat_copy()*
7. *glm_quat_from_vecs()*
8. *glm_quat_norm()*
9. *glm_quat_normalize()*
10. *glm_quat_normalize_to()*
11. *glm_quat_dot()*
12. *glm_quat_conjugate()*
13. *glm_quat_inv()*
14. *glm_quat_add()*
15. *glm_quat_sub()*
16. *glm_quat_real()*
17. *glm_quat_imag()*
18. *glm_quat_imagn()*
19. *glm_quat_imaglen()*
20. *glm_quat_angle()*
21. *glm_quat_axis()*
22. *glm_quat_mul()*
23. *glm_quat_mat4()*
24. *glm_quat_mat4t()*
25. *glm_quat_mat3()*
26. *glm_quat_mat3t()*
27. *glm_quat_lerp()*
28. *glm_quat_nlerp()*
29. *glm_quat_slerp()*
30. *glm_quat_look()*
31. *glm_quat_for()*
32. *glm_quat_forp()*
33. *glm_quat_rotatev()*
34. *glm_quat_rotate()*
35. *glm_quat_rotate_at()*
36. *glm_quat_rotate_atm()*

5.7.2 Functions documentation

void **glm_quat_identity** (versor *q*)

makes given quat to identity

Parameters:

[in, out] **q** quaternion

void **glm_quat_identity_array** (versor * __restrict *q*, size_t *count*)

make given quaternion array's each element identity quaternion

Parameters:

[in, out] **q** quat array (must be aligned (16) if alignment is not disabled)
[in] **count** count of quaternions

void **glm_quat_init** (versor *q*, float *x*, float *y*, float *z*, float *w*)

inits quaternion with given values

Parameters:

[out] **q** quaternion
[in] **x** imag.x
[in] **y** imag.y
[in] **z** imag.z
[in] **w** w (real part)

void **glm_quat** (versor *q*, float *angle*, float *x*, float *y*, float *z*)

creates NEW quaternion with individual axis components

given axis will be normalized

Parameters:

[out] **q** quaternion
[in] **angle** angle (radians)
[in] **x** axis.x
[in] **y** axis.y
[in] **z** axis.z

void **glm_quatv** (versor *q*, float *angle*, vec3 *axis*)

creates NEW quaternion with axis vector

given axis will be normalized

Parameters:

[out] **q** quaternion
[in] **angle** angle (radians)
[in] **axis** axis (will be normalized)

void **glm_quat_copy** (versor *q*, versor *dest*)

copy quaternion to another one

Parameters:

[in] **q** source quaternion
[out] **dest** destination quaternion

void **glm_quat_from_vecs** (vec3 *a*, vec3 *b*, versor *dest*)

compute unit quaternion needed to rotate a into b

References:

- Finding quaternion representing the rotation from one vector to another
- Quaternion from two vectors
- Angle between vectors

Parameters:

[in] **a** unit vector
[in] **b** unit vector
[in] **dest** unit quaternion

float **glm_quat_norm** (versor *q*)

returns norm (magnitude) of quaternion

Parameters:

[in] **a** quaternion

Returns: norm (magnitude)

void **glm_quat_normalize_to** (versor *q*, versor *dest*)

normalize quaternion and store result in dest, original one will not be normalized

Parameters:

[in] **q** quaternion to normalize into
[out] **dest** destination quaternion

void **glm_quat_normalize** (versor *q*)

normalize quaternion

Parameters:

[in, out] **q** quaternion

float **glm_quat_dot** (versor *p*, versor *q*)
dot product of two quaternion

Parameters:

[in] **p** quaternion 1

[in] **q** quaternion 2

Returns: dot product

void **glm_quat_conjugate** (versor *q*, versor *dest*)
conjugate of quaternion

Parameters:

[in] **q** quaternion

[in] **dest** conjugate

void **glm_quat_inv** (versor *q*, versor *dest*)
inverse of non-zero quaternion

Parameters:

[in] **q** quaternion

[in] **dest** inverse quaternion

void **glm_quat_add** (versor *p*, versor *q*, versor *dest*)
add (componentwise) two quaternions and store result in dest

Parameters:

[in] **p** quaternion 1

[in] **q** quaternion 2

[in] **dest** result quaternion

void **glm_quat_sub** (versor *p*, versor *q*, versor *dest*)
subtract (componentwise) two quaternions and store result in dest

Parameters:

[in] **p** quaternion 1

[in] **q** quaternion 2

[in] **dest** result quaternion

float **glm_quat_real** (versor *q*)
returns real part of quaternion

Parameters:

[in] **q** quaternion

Returns: real part (quat.w)

void **glm_quat_imag** (versor *q*, vec3 *dest*)
returns imaginary part of quaternion

Parameters:

[in] **q** quaternion

[out] **dest** imag

void **glm_quat_imagn**(versor *q*, vec3 *dest*)

returns normalized imaginary part of quaternion

Parameters:

[in] **q** quaternion

[out] **dest** imag

float **glm_quat_imaglen**(versor *q*)

returns length of imaginary part of quaternion

Parameters:

[in] **q** quaternion

Returns: norm of imaginary part

float **glm_quat_angle**(versor *q*)

returns angle of quaternion

Parameters:

[in] **q** quaternion

Returns: angles of quat (radians)

void **glm_quat_axis**(versor *q*, versor *dest*)

axis of quaternion

Parameters:

[in] **p** quaternion

[out] **dest** axis of quaternion

void **glm_quat_mul**(versor *p*, versor *q*, versor *dest*)

multiplies two quaternion and stores result in dest

this is also called Hamilton Product

According to WikiPedia:
The product of two rotation quaternions [clarification needed] will be equivalent to the rotation q followed by the rotation p

Parameters:

[in] **p** quaternion 1 (first rotation)

[in] **q** quaternion 2 (second rotation)

[out] **dest** result quaternion

void **glm_quat_mat4**(versor *q*, mat4 *dest*)

convert quaternion to mat4

Parameters:

[in] **q** quaternion
[out] **dest** result matrix

void **glm_quat_mat4t** (versor *q*, mat4 *dest*)

convert quaternion to mat4 (transposed). This is transposed version of glm_quat_mat4

Parameters:

[in] **q** quaternion
[out] **dest** result matrix

void **glm_quat_mat3** (versor *q*, mat3 *dest*)

convert quaternion to mat3

Parameters:

[in] **q** quaternion
[out] **dest** result matrix

void **glm_quat_mat3t** (versor *q*, mat3 *dest*)

convert quaternion to mat3 (transposed). This is transposed version of glm_quat_mat3

Parameters:

[in] **q** quaternion
[out] **dest** result matrix

void **glm_quat_lerp** (versor *from*, versor *to*, float *t*, versor *dest*)

interpolates between two quaternions
using spherical linear interpolation (LERP)

Parameters:

[in] **from** from
[in] **to** to
[in] **t** interpolant (amount) clamped between 0 and 1
[out] **dest** result quaternion

void **glm_quat_nlerp** (versor *q*, versor *r*, float *t*, versor *dest*)

interpolates between two quaternions
taking the shortest rotation path using
normalized linear interpolation (NLERP)

This is a cheaper alternative to slerp; most games use nlerp for animations as it visually makes little difference.

References:

- Understanding Slerp, Then Not Using it
- Lerp, Slerp and Nlerp

Parameters:

[in] **from** from
[in] **to** to
[in] **t** interpolant (amount) clamped between 0 and 1
[out] **dest** result quaternion

void **glm_quat_slerp** (versor *q*, versor *r*, float *t*, versor *dest*)

interpolates between two quaternions
using spherical linear interpolation (SLERP)

Parameters:

[in] **from** from
[in] **to** to
[in] **t** interpolant (amount) clamped between 0 and 1
[out] **dest** result quaternion

void **glm_quat_look** (vec3 *eye*, versor *ori*, mat4 *dest*)

creates view matrix using quaternion as camera orientation

Parameters:

[in] **eye** eye
[in] **ori** orientation in world space as quaternion
[out] **dest** result matrix

void **glm_quat_for** (vec3 *dir*, vec3 *up*, versor *dest*)

creates look rotation quaternion

Parameters:

[in] **dir** direction to look
[in] **up** up vector
[out] **dest** result matrix

void **glm_quat_forp** (vec3 *from*, vec3 *to*, vec3 *up*, versor *dest*)

creates look rotation quaternion using source and destination positions p suffix stands for position

this is similar to glm_quat_for except this computes direction for glm_quat_for for you.

Parameters:

[in] **from** source point
[in] **to** destination point
[in] **up** up vector
[out] **dest** result matrix

void **`glm_quat_rotatev`** (versor *q*, vec3 *v*, vec3 *dest*)

crotate vector using using quaternion

Parameters:

[in] **q** quaternion
[in] **v** vector to rotate
[out] **dest** rotated vector

void **`glm_quat_rotate`** (mat4 *m*, versor *q*, mat4 *dest*)

rotate existing transform matrix using quaternion

instead of passing identity matrix, consider to use quat_mat4 functions

Parameters:

[in] **m** existing transform matrix to rotate
[in] **q** quaternion
[out] **dest** rotated matrix/transform

void **`glm_quat_rotate_at`** (mat4 *m*, versor *q*, vec3 *pivot*)

rotate existing transform matrix using quaternion at pivot point

Parameters:

[in, out] **m** existing transform matrix to rotate
[in] **q** quaternion
[in] **pivot** pivot

void **`glm_quat_rotate_atm`** (mat4 *m*, versor *q*, vec3 *pivot*)

rotate NEW transform matrix using quaternion at pivot point
this creates rotation matrix, it assumes you don't have a matrix

this should work faster than glm_quat_rotate_at because it reduces one glm_translate.

Parameters:

[in, out] **m** existing transform matrix to rotate
[in] **q** quaternion
[in] **pivot** pivot

5.8 euler angles

Header: cglm/euler.h

You may wonder what **glm_euler_sq** type (**_sq** stands for sequence) and *glm_euler_by_order()* do. I used them to convert euler angles in one coordinate system to another. For instance if you have **Z_UP** euler angles and if you want to convert it to **Y_UP** axis then *glm_euler_by_order()* is your friend. For more information check *glm_euler_order()* documentation

You must pass arrays as array, if you use C compiler then you can use something like this:

```
float pitch, yaw, roll;
mat4  rot;

/* pitch = ...; yaw = ...; roll = ... */
glm_euler((vec3){pitch, yaw, roll}, rot);
```

5.8.1 Rotation Conveniention

Current *cglm*'s euler functions uses these convention:

- Tait–Bryan angles (x-y-z convention)
- Intrinsic rotations (pitch, yaw and roll). This is reserve order of extrinsic (elevation, heading and bank) rotation
- Right hand rule (actually all rotations in *cglm* use **RH**)
- All angles used in *cglm* are **RADIANS** not degrees

NOTE: The default *glm_euler()* function is the short name of *glm_euler_xyz()* this is why you can't see *glm_euler_xyz()*. When you see an euler function which doesn't have any X, Y, Z suffix then assume that uses **_xyz** (or instead it accept order as parameter).

If rotation doesn't work properly, your options:

1. If you use (or paste) degrees convert it to radians before calling an euler function

```
float pitch, yaw, roll;
mat4  rot;

/* pitch = degrees; yaw = degrees; roll = degrees */
glm_euler((vec3){glm_rad(pitch), glm_rad(yaw), glm_rad(roll)}, rot);
```

2. Convention mismatch. You may have extrinsic angles, if you do (if you must) then consider to use reverse order e.g if you have **xyz** extrinsic then use **zyx**
3. *cglm* may implemented it wrong, consider to create an issue to report it or pull request to fix it

5.8.2 Table of contents (click to go):

Types:

Functions:

3. *glm_euler()*
4. *glm_euler_xyz()*
5. *glm_euler_zyx()*
6. *glm_euler_zxy()*
7. *glm_euler_xzy()*
8. *glm_euler_yzx()*
9. *glm_euler_yxz()*
10. *glm_euler_by_order()*

5.8.3 Functions documentation

glm_euler_sq **glm_euler_order** (int *ord[3]*)

packs euler angles order to glm_euler_sq enum.

To use *glm_euler_by_order()* function you need *glm_euler_sq*. You can get it with this function.

You can build param like this:

X = 0, Y = 1, Z = 2

if you have ZYX order then you pass this: [2, 1, 0] = ZYX. if you have YXZ order then you pass this: [1, 0, 2] = YXZ

As you can see first item specifies which axis will be first then the second one specifies which one will be next an so on.

Parameters:

[in] **ord** euler angles order [Angle1, Angle2, Angle2]

Returns: packed euler order

void **glm_euler_angles** (mat4 *m*, vec3 *dest*)

extract euler angles (in radians) using xyz order

Parameters:

[in] **m** affine transform

[out] **dest** angles vector [x, y, z]

void **glm_euler** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

this is alias of glm_euler_xyz function

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]

[in] **dest** rotation matrix

void **glm_euler_xyz** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_zyx** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_zxy** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_xzy** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_yzx** (vec3 *angles*, mat4 *dest*)
build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_yxz** (vec3 *angles*, mat4 *dest*)

build rotation matrix from euler angles

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **dest** rotation matrix

void **glm_euler_by_order**(vec3 *angles*, glm_euler_sq *ord*, mat4 *dest*)

build rotation matrix from euler angles with given euler order.

Use *glm_euler_order()* function to build *ord* parameter

Parameters:

[in] **angles** angles as vector [Xangle, Yangle, Zangle]
[in] **ord** euler order
[in] **dest** rotation matrix

5.9 mat4

Header: cglm/mat4.h

Important: *glm_mat4_scale()* multiplies mat4 with scalar, if you need to apply scale transform use *glm_scale()* functions.

5.9.1 Table of contents (click to go):

Macros:

1. GLM_MAT4_IDENTITY_INIT
2. GLM_MAT4_ZERO_INIT
3. GLM_MAT4_IDENTITY
4. GLM_MAT4_ZERO
5. glm_mat4_udup(mat, dest)
6. glm_mat4_dup(mat, dest)

Functions:

14. glm_mat3_trace3()
15. *glm_mat4_quat()*
16. *glm_mat4_transpose_to()*
17. *glm_mat4_transpose()*
18. *glm_mat4_scale_p()*
19. *glm_mat4_scale()*
20. *glm_mat4_det()*
21. *glm_mat4_inv()*
22. *glm_mat4_inv_fast()*
23. *glm_mat4_swap_col()*
24. *glm_mat4_swap_row()*
25. *glm_mat4_rmc()*

5.9.2 Functions documentation

void **glm_mat4_ucopy**(mat4 *mat*, mat4 *dest*)
copy mat4 to another one (dest). u means align is not required for dest

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat4_copy**(mat4 *mat*, mat4 *dest*)
copy mat4 to another one (dest).

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat4_identity**(mat4 *mat*)
copy identity mat4 to mat, or makes mat to identiy

Parameters:

[out] **mat** matrix

void **glm_mat4_identity_array**(mat4 * __restrict *mat*, size_t *count*)
make given matrix array's each element identity matrix

Parameters:

[in,out] **mat** matrix array (must be aligned (16/32) if alignment is not disabled)
[in] **count** count of matrices

void **glm_mat4_zero**(mat4 *mat*)
make given matrix zero

Parameters:

[in,out] **mat** matrix to

void **glm_mat4_pick3**(mat4 *mat*, mat3 *dest*)
copy upper-left of mat4 to mat3

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat4_pick3t**(mat4 *mat*, mat4 *dest*)

copy upper-left of mat4 to mat3 (transposed) the postfix t stands for transpose

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat4_ins3**(mat3 *mat*, mat4 *dest*)

copy mat3 to mat4's upper-left. this function does not fill mat4's other elements. To do that use glm_mat4.

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat4_mul**(mat4 *m1*, mat4 *m2*, mat4 *dest*)

multiply m1 and m2 to dest m1, m2 and dest matrices can be same matrix, it is possible to write this:

```
mat4 m = GLM_MAT4_IDENTITY_INIT;
glm_mat4_mul(m, m, m);
```

Parameters:

[in] **m1** left matrix

[in] **m2** right matrix

[out] **dest** destination matrix

void **glm_mat4_mulN**(mat4 * __restrict *matrices[]*, int *len*, mat4 *dest*)

mupliply N mat4 matrices and store result in dest | this function lets you multiply multiple (more than two or more…) | matrices

multiplication will be done in loop, this may reduce instructions

size but if **len** is too small then compiler may unroll whole loop

```
mat m1, m2, m3, m4, res;
glm_mat4_mulN((mat4 *[]){&m1, &m2, &m3, &m4}, 4, res);
```

Parameters:

[in] **matrices** array of mat4

[in] **len** matrices count

[out] **dest** destination matrix

void **glm_mat4_mulv**(mat4 *m*, vec4 *v*, vec4 *dest*)

multiply mat4 with vec4 (column vector) and store in dest vector

Parameters:

[in] **m** mat4 (left)

[in] **v** vec4 (right, column vector)

[in] **last** 4th item to make it vec4

[out] **dest** vec4 (result, column vector)

void **glm_mat4_mulv3** (mat4 *m*, vec3 *v*, float *last*, vec3 *dest*)

multiply **vec3** with **mat4** and get **vec3** as result

actually the result is **vec4**, after multiplication, the last component is trimmed, if you need the result's last component then don't use this function and consider to use **glm_mat4_mulv()**

Parameters:

[in] **m** mat4(affine transform)

[in] **v** vec3

[in] **last** 4th item to make it vec4

[out] **dest** result vector (vec3)

void **glm_mat4_trace** (mat4 *m*)

sum of the elements on the main diagonal from upper left to the lower right

Parameters:

[in] **m** matrix

Returns: trace of matrix

void **glm_mat4_trace3** (mat4 *m*)

trace of matrix (rotation part)

sum of the elements on the main diagonal from upper left to the lower right

Parameters:

[in] **m** matrix

Returns: trace of matrix

void **glm_mat4_quat** (mat4 *m*, versor *dest*)

convert mat4's rotation part to quaternion

Parameters: | *[in]* **m** affine matrix | *[out]* **dest** destination quaternion

void **glm_mat4_transpose_to** (mat4 *m*, mat4 *dest*)

transpose mat4 and store in dest source matrix will not be transposed unless dest is m

Parameters:

[in] **m** matrix

[out] **dest** destination matrix

void **glm_mat4_transpose** (mat4 *m*)

tranpose mat4 and store result in same matrix

Parameters:

[in] **m** source

[out] **dest** destination matrix

void **glm_mat4_scale_p**(mat4 *m*, float *s*)
scale (multiply with scalar) matrix without simd optimization

Parameters:

[in, out] **m** matrix
[in] **s** scalar

void **glm_mat4_scale**(mat4 *m*, float *s*)
scale (multiply with scalar) matrix THIS IS NOT SCALE TRANSFORM, use glm_scale for that.

Parameters:

[in, out] **m** matrix
[in] **s** scalar

float **glm_mat4_det**(mat4 *mat*)
mat4 determinant

Parameters:

[in] **mat** matrix

Return:

determinant

void **glm_mat4_inv**(mat4 *mat*, mat4 *dest*)
inverse mat4 and store in dest

Parameters:

[in] **mat** source
[out] **dest** destination matrix (inverse matrix)

void **glm_mat4_inv_fast**(mat4 *mat*, mat4 *dest*)
inverse mat4 and store in dest

this func uses reciprocal approximation without extra corrections
e.g Newton-Raphson. this should work faster than normal,
to get more precise use glm_mat4_inv version.

NOTE: You will lose precision, glm_mat4_inv is more accurate

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat4_swap_col**(mat4 *mat*, int *col1*, int *col2*)
swap two matrix columns

Parameters:

[in, out] **mat** matrix
[in] **col1** col1
[in] **col2** col2

void **glm_mat4_swap_row** (mat4 *mat*, int *row1*, int *row2*)
swap two matrix rows

Parameters:

- *[in, out]* **mat** matrix
- *[in]* **row1** row1
- *[in]* **row2** row2

float **glm_mat4_rmc** (vec4 *r*, mat4 *m*, vec4 *c*)

rmc stands for **Row** * **Matrix** * **Column**

helper for R (row vector) * M (matrix) * C (column vector)

the result is scalar because R * M = Matrix1x4 (row vector),
then Matrix1x4 * Vec4 (column vector) = Matrix1x1 (Scalar)

Parameters:

- *[in]* **r** row vector or matrix1x4
- *[in]* **m** matrix4x4
- *[in]* **c** column vector or matrix4x1

Returns: scalar value e.g. Matrix1x1

5.10 mat3

Header: cglm/mat3.h

5.10.1 Table of contents (click to go):

Macros:

1. GLM_MAT3_IDENTITY_INIT
2. GLM_MAT3_ZERO_INIT
3. GLM_MAT3_IDENTITY
4. GLM_MAT3_ZERO
5. glm_mat3_dup(mat, dest)

Functions:

1. *glm_mat3_copy()*
2. *glm_mat3_identity()*
3. *glm_mat3_identity_array()*
4. *glm_mat3_zero()*
5. *glm_mat3_mul()*

6. *glm_mat3_transpose_to()*
7. *glm_mat3_transpose()*
8. *glm_mat3_mulv()*
9. *glm_mat3_quat()*
10. *glm_mat3_scale()*
11. *glm_mat3_det()*
12. *glm_mat3_inv()*
13. *glm_mat3_trace()*
14. *glm_mat3_swap_col()*
15. *glm_mat3_swap_row()*
16. *glm_mat3_rmc()*

5.10.2 Functions documentation

void **glm_mat3_copy**(mat3 *mat*, mat3 *dest*)

copy mat3 to another one (dest).

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat3_identity**(mat3 *mat*)

copy identity mat3 to mat, or makes mat to identiy

Parameters:

[out] **mat** matrix

void **glm_mat3_identity_array**(mat3 * __restrict *mat*, size_t *count*)

make given matrix array's each element identity matrix

Parameters:

[in,out] **mat** matrix array (must be aligned (16/32) if alignment is not disabled)

[in] **count** count of matrices

void **glm_mat3_zero**(mat3 *mat*)

make given matrix zero

Parameters:

[in,out] **mat** matrix to

void **glm_mat3_mul**(mat3 *m1*, mat3 *m2*, mat3 *dest*)

multiply m1 and m2 to dest m1, m2 and dest matrices can be same matrix, it is possible to write this:

```
mat3 m = GLM_MAT3_IDENTITY_INIT;
glm_mat3_mul(m, m, m);
```

Parameters:

[in] **m1** left matrix

[in] **m2** right matrix

[out] **dest** destination matrix

void **glm_mat3_transpose_to**(mat3 *m*, mat3 *dest*)
transpose mat4 and store in dest source matrix will not be transposed unless dest is m

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat3_transpose**(mat3 *m*)
tranpose mat3 and store result in same matrix

Parameters:

[in] **mat** source

[out] **dest** destination

void **glm_mat3_mulv**(mat3 *m*, vec3 *v*, vec3 *dest*)
multiply mat4 with vec4 (column vector) and store in dest vector

Parameters:

[in] **mat** mat3 (left)

[in] **v** vec3 (right, column vector)

[out] **dest** destination (result, column vector)

void **glm_mat3_quat**(mat3 *m*, versor *dest*)
convert mat3 to quaternion

Parameters:

[in] **m** rotation matrix

[out] **dest** destination quaternion

void **glm_mat3_scale**(mat3 *m*, float *s*)
multiply matrix with scalar

Parameters:

[in, out] **mat** matrix

[in] **dest** scalar

float **glm_mat3_det**(mat3 *mat*)
returns mat3 determinant

Parameters:

[in] **mat** matrix

Returns: mat3 determinant

void **glm_mat3_inv**(mat3 *mat*, mat3 *dest*)
inverse mat3 and store in dest

Parameters:

[in] **mat** matrix

[out] **dest** destination (inverse matrix)

void **glm_mat3_trace**(mat3 *m*)

sum of the elements on the main diagonal from upper left to the lower right

Parameters:

[in] **m** matrix

Returns: trace of matrix

void **glm_mat3_swap_col**(mat3 *mat*, int *col1*, int *col2*)
swap two matrix columns

Parameters:

[in, out] **mat** matrix
[in] **col1** col1
[in] **col2** col2

void **glm_mat3_swap_row**(mat3 *mat*, int *row1*, int *row2*)
swap two matrix rows

Parameters:

[in, out] **mat** matrix
[in] **row1** row1
[in] **row2** row2

float **glm_mat3_rmc**(vec3 *r*, mat3 *m*, vec3 *c*)

rmc stands for **Row** * **Matrix** * **Column**

helper for R (row vector) * M (matrix) * C (column vector)

the result is scalar because R * M = Matrix1x3 (row vector),
then Matrix1x3 * Vec3 (column vector) = Matrix1x1 (Scalar)

Parameters:

[in] **r** row vector or matrix1x3
[in] **m** matrix3x3
[in] **c** column vector or matrix3x1

Returns: scalar value e.g. Matrix1x1

5.11 mat2

Header: cglm/mat2.h

5.11.1 Table of contents (click to go):

Macros:

1. GLM_mat2_IDENTITY_INIT
2. GLM_mat2_ZERO_INIT
3. GLM_mat2_IDENTITY

4. GLM_mat2_ZERO

Functions:

1. *glm_mat2_copy()*
2. *glm_mat2_identity()*
3. *glm_mat2_identity_array()*
4. *glm_mat2_zero()*
5. *glm_mat2_mul()*
6. *glm_mat2_transpose_to()*
7. *glm_mat2_transpose()*
8. *glm_mat2_mulv()*
9. *glm_mat2_scale()*
10. *glm_mat2_det()*
11. *glm_mat2_inv()*
12. *glm_mat2_trace()*
13. *glm_mat2_swap_col()*
14. *glm_mat2_swap_row()*
15. *glm_mat2_rmc()*

5.11.2 Functions documentation

void **glm_mat2_copy**(mat2 *mat*, mat2 *dest*)
copy mat2 to another one (dest).

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat2_identity**(mat2 *mat*)
copy identity mat2 to mat, or makes mat to identiy

Parameters:

[out] **mat** matrix

void **glm_mat2_identity_array**(mat2 * __restrict *mat*, size_t *count*)
make given matrix array's each element identity matrix

Parameters:

[in,out] **mat** matrix array (must be aligned (16/32) if alignment is not disabled)
[in] **count** count of matrices

void **glm_mat2_zero**(mat2 *mat*)
make given matrix zero

Parameters:

[in,out] **mat** matrix to

void **glm_mat2_mul** (mat2 *m1*, mat2 *m2*, mat2 *dest*)
multiply m1 and m2 to dest m1, m2 and dest matrices can be same matrix, it is possible to write this:

```
mat2 m = GLM_mat2_IDENTITY_INIT;
glm_mat2_mul(m, m, m);
```

Parameters:

[in] **m1** left matrix
[in] **m2** right matrix
[out] **dest** destination matrix

void **glm_mat2_transpose_to** (mat2 *m*, mat2 *dest*)
transpose mat4 and store in dest source matrix will not be transposed unless dest is m

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat2_transpose** (mat2 *m*)
tranpose mat2 and store result in same matrix

Parameters:

[in] **mat** source
[out] **dest** destination

void **glm_mat2_mulv** (mat2 *m*, vec2 *v*, vec2 *dest*)
multiply mat4 with vec4 (column vector) and store in dest vector

Parameters:

[in] **mat** mat2 (left)
[in] **v** vec2 (right, column vector)
[out] **dest** destination (result, column vector)

void **glm_mat2_scale** (mat2 *m*, float *s*)
multiply matrix with scalar

Parameters:

[in, out] **mat** matrix
[in] **dest** scalar

float **glm_mat2_det** (mat2 *mat*)
returns mat2 determinant

Parameters:

[in] **mat** matrix

Returns: mat2 determinant

void **glm_mat2_inv** (mat2 *mat*, mat2 *dest*)
inverse mat2 and store in dest

Parameters:

[in] **mat** matrix
[out] **dest** destination (inverse matrix)

void **glm_mat2_trace**(mat2 *m*)

sum of the elements on the main diagonal from upper left to the lower right

Parameters:

[in] **m** matrix

Returns: trace of matrix

void **glm_mat2_swap_col**(mat2 *mat*, int *col1*, int *col2*)
swap two matrix columns

Parameters:

[in, out] **mat** matrix
[in] **col1** col1
[in] **col2** col2

void **glm_mat2_swap_row**(mat2 *mat*, int *row1*, int *row2*)
swap two matrix rows

Parameters:

[in, out] **mat** matrix
[in] **row1** row1
[in] **row2** row2

float **glm_mat2_rmc**(vec2 *r*, mat2 *m*, vec2 *c*)

rmc stands for **Row** * **Matrix** * **Column**

helper for R (row vector) * M (matrix) * C (column vector)

the result is scalar because R * M = Matrix1x2 (row vector),
then Matrix1x2 * Vec2 (column vector) = Matrix1x1 (Scalar)

Parameters:

[in] **r** row vector or matrix1x2
[in] **m** matrix2x2
[in] **c** column vector or matrix2x1

Returns: scalar value e.g. Matrix1x1

5.12 vec3

Header: cglm/vec3.h

Important: *cglm* was used **glm_vec_** namespace for vec3 functions until **v0.5.0**, since **v0.5.0** cglm uses **glm_vec3_** namespace for vec3.

Also *glm_vec3_flipsign* has been renamed to *glm_vec3_negate*

We mostly use vectors in graphics math, to make writing code faster and easy to read, some *vec3* functions are aliased in global namespace. For instance `glm_dot()` is alias of *glm_vec3_dot()*, alias means inline wrapper here. There is no call verison of alias functions

There are also functions for rotating *vec3* vector. **_m4**, **_m3** prefixes rotate *vec3* with matrix.

5.12.1 Table of contents (click to go):

Macros:

1. glm_vec3_dup(v, dest)
2. GLM_VEC3_ONE_INIT
3. GLM_VEC3_ZERO_INIT
4. GLM_VEC3_ONE
5. GLM_VEC3_ZERO
6. GLM_YUP
7. GLM_ZUP
8. GLM_XUP

Functions:

1. `glm_vec3()`
2. `glm_vec3_copy()`
3. `glm_vec3_zero()`
4. `glm_vec3_one()`
5. `glm_vec3_dot()`
6. `glm_vec3_norm2()`
7. `glm_vec3_norm()`
8. `glm_vec3_add()`
9. `glm_vec3_adds()`
10. `glm_vec3_sub()`
11. `glm_vec3_subs()`
12. `glm_vec3_mul()`
13. `glm_vec3_scale()`
14. `glm_vec3_scale_as()`
15. `glm_vec3_div()`
16. `glm_vec3_divs()`
17. `glm_vec3_addadd()`
18. `glm_vec3_subadd()`
19. `glm_vec3_muladd()`
20. `glm_vec3_muladds()`
21. `glm_vec3_maxadd()`

22. *glm_vec3_minadd()*
23. *glm_vec3_flipsign()*
24. *glm_vec3_flipsign_to()*
25. *glm_vec3_inv()*
26. *glm_vec3_inv_to()*
27. *glm_vec3_negate()*
28. *glm_vec3_negate_to()*
29. *glm_vec3_normalize()*
30. *glm_vec3_normalize_to()*
31. *glm_vec3_cross()*
32. *glm_vec3_crossn()*
33. *glm_vec3_distance2()*
34. *glm_vec3_distance()*
35. *glm_vec3_angle()*
36. *glm_vec3_rotate()*
37. *glm_vec3_rotate_m4()*
38. *glm_vec3_rotate_m3()*
39. *glm_vec3_proj()*
40. *glm_vec3_center()*
41. *glm_vec3_maxv()*
42. *glm_vec3_minv()*
43. *glm_vec3_ortho()*
44. *glm_vec3_clamp()*
45. *glm_vec3_lerp()*

5.12.2 Functions documentation

void **glm_vec3** (vec4 *v4*, vec3 *dest*)
init vec3 using vec4

Parameters:

[in] **v4** vector4
[out] **dest** destination

void **glm_vec3_copy** (vec3 *a*, vec3 *dest*)
copy all members of [a] to [dest]

Parameters:

[in] **a** source
[out] **dest** destination

void **glm_vec3_zero**(vec3 *v*)
makes all members 0.0f (zero)

Parameters:

[in, out] **v** vector

void **glm_vec3_one**(vec3 *v*)
makes all members 1.0f (one)

Parameters:

[in, out] **v** vector

float **glm_vec3_dot**(vec3 *a*, vec3 *b*)
dot product of vec3

Parameters:

[in] **a** vector1

[in] **b** vector2

Returns: dot product

void **glm_vec3_cross**(vec3 *a*, vec3 *b*, vec3 *d*)
cross product of two vector (RH)

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** destination

void **glm_vec3_crossn**(vec3 *a*, vec3 *b*, vec3 *dest*)
cross product of two vector (RH) and normalize the result

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** destination

float **glm_vec3_norm2**(vec3 *v*)
norm * norm (magnitude) of vector

we can use this func instead of calling norm * norm, because it would call sqrtf fuction twice but with this func we can avoid func call, maybe this is not good name for this func

Parameters:

[in] **v** vector

Returns: square of norm / magnitude

float **glm_vec3_norm**(vec3 *vec*)

euclidean norm (magnitude), also called L2 norm
this will give magnitude of vector in euclidean space

Parameters:

[in] **vec** vector

void **glm_vec3_add**(vec3 *a*, vec3 *b*, vec3 *dest*)
add a vector to b vector store result in dest

Parameters:

- *[in]* **a** vector1
- *[in]* **b** vector2
- *[out]* **dest** destination vector

void **glm_vec3_adds**(vec3 *a*, float *s*, vec3 *dest*)
add scalar to v vector store result in dest (d = v + vec(s))

Parameters:

- *[in]* **v** vector
- *[in]* **s** scalar
- *[out]* **dest** destination vector

void **glm_vec3_sub**(vec3 *v1*, vec3 *v2*, vec3 *dest*)
subtract b vector from a vector store result in dest (d = v1 - v2)

Parameters:

- *[in]* **a** vector1
- *[in]* **b** vector2
- *[out]* **dest** destination vector

void **glm_vec3_subs**(vec3 *v*, float *s*, vec3 *dest*)
subtract scalar from v vector store result in dest (d = v - vec(s))

Parameters:

- *[in]* **v** vector
- *[in]* **s** scalar
- *[out]* **dest** destination vector

void **glm_vec3_mul**(vec3 *a*, vec3 *b*, vec3 *d*)
multiply two vector (component-wise multiplication)

Parameters:

- *[in]* **a** vector
- *[in]* **b** scalar
- *[out]* **d** result = (a[0] * b[0], a[1] * b[1], a[2] * b[2])

void **glm_vec3_scale**(vec3 *v*, float *s*, vec3 *dest*)

multiply/scale vec3 vector with scalar: result = v * s

Parameters:

- *[in]* **v** vector
- *[in]* **s** scalar
- *[out]* **dest** destination vector

void **glm_vec3_scale_as**(vec3 *v*, float *s*, vec3 *dest*)
make vec3 vector scale as specified: result = unit(v) * s

Parameters:

- *[in]* **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **`glm_vec3_div`**(vec3 *a*, vec3 *b*, vec3 *dest*)

div vector with another component-wise division: d = a / b

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** result = (a[0] / b[0], a[1] / b[1], a[2] / b[2])

void **`glm_vec3_divs`**(vec3 *v*, float *s*, vec3 *dest*)

div vector with scalar: d = v / s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** result = (a[0] / s, a[1] / s, a[2] / s])

void **`glm_vec3_addadd`**(vec3 *a*, vec3 *b*, vec3 *dest*)

add two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a + b)

void **`glm_vec3_subadd`**(vec3 *a*, vec3 *b*, vec3 *dest*)

sub two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a - b)

void **`glm_vec3_muladd`**(vec3 *a*, vec3 *b*, vec3 *dest*)

mul two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a * b)

void **glm_vec3_muladds** (vec3 *a*, float *s*, vec3 *dest*)

mul vector with scalar and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector
[in] **s** scalar
[out] **dest** dest += (a * b)

void **glm_vec3_maxadd** (vec3 *a*, vec3 *b*, vec3 *dest*)

add max of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a * b)

void **glm_vec3_minadd** (vec3 *a*, vec3 *b*, vec3 *dest*)

add min of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a * b)

void **glm_vec3_flipsign** (vec3 *v*)

DEPRACATED!

use `glm_vec3_negate()`

Parameters:

[in, out] **v** vector

void **glm_vec3_flipsign_to** (vec3 *v*, vec3 *dest*)

DEPRACATED!

use `glm_vec3_negate_to()`

Parameters:

[in] **v** vector
[out] **dest** negated vector

void **glm_vec3_inv** (vec3 *v*)

DEPRACATED!

use `glm_vec3_negate()`

Parameters:

[in, out] **v** vector

void **glm_vec3_inv_to**(vec3 *v*, vec3 *dest*)
DEPRACATED!

use *glm_vec3_negate_to()*

Parameters:

[in] **v** source

[out] **dest** destination

void **glm_vec3_negate**(vec3 *v*)
negate vector components

Parameters:

[in, out] **v** vector

void **glm_vec3_negate_to**(vec3 *v*, vec3 *dest*)
negate vector components and store result in dest

Parameters:

[in] **v** vector

[out] **dest** negated vector

void **glm_vec3_normalize**(vec3 *v*)
normalize vec3 and store result in same vec

Parameters:

[in, out] **v** vector

void **glm_vec3_normalize_to**(vec3 *vec*, vec3 *dest*)

normalize vec3 to dest

Parameters:

[in] **vec** source

[out] **dest** destination

float **glm_vec3_angle**(vec3 *v1*, vec3 *v2*)
angle betwen two vector

Parameters:

[in] **v1** vector1

[in] **v2** vector2

Return:

angle as radians

void **glm_vec3_rotate**(vec3 *v*, float *angle*, vec3 *axis*)

rotate vec3 around axis by angle using Rodrigues' rotation formula

Parameters:

[in, out] **v** vector

[in] **axis** axis vector (will be normalized)

[in] **angle** angle (radians)

void **glm_vec3_rotate_m4** (mat4 *m*, vec3 *v*, vec3 *dest*)
apply rotation matrix to vector

Parameters:

[in] **m** affine matrix or rot matrix
[in] **v** vector
[out] **dest** rotated vector

void **glm_vec3_rotate_m3** (mat3 *m*, vec3 *v*, vec3 *dest*)
apply rotation matrix to vector

Parameters:

[in] **m** affine matrix or rot matrix
[in] **v** vector
[out] **dest** rotated vector

void **glm_vec3_proj** (vec3 *a*, vec3 *b*, vec3 *dest*)
project a vector onto b vector

Parameters:

[in] **a** vector1
[in] **b** vector2
[out] **dest** projected vector

void **glm_vec3_center** (vec3 *v1*, vec3 *v2*, vec3 *dest*)
find center point of two vector

Parameters:

[in] **v1** vector1
[in] **v2** vector2
[out] **dest** center point

float **glm_vec3_distance2** (vec3 *v1*, vec3 *v2*)
squared distance between two vectors

Parameters:

[in] **v1** vector1
[in] **v2** vector2

Returns:

squared distance (distance * distance)

float **glm_vec3_distance** (vec3 *v1*, vec3 *v2*)
distance between two vectors

Parameters:

[in] **v1** vector1
[in] **v2** vector2

Returns:

distance

void **glm_vec3_maxv** (vec3 *v1*, vec3 *v2*, vec3 *dest*)
max values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec3_minv** (vec3 *v1*, vec3 *v2*, vec3 *dest*)
min values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec3_ortho** (vec3 *v*, vec3 *dest*)
possible orthogonal/perpendicular vector

References:

- On picking an orthogonal vector (and combing coconuts)

Parameters:

[in] **v** vector

[out] **dest** orthogonal/perpendicular vector

void **glm_vec3_clamp** (vec3 *v*, float *minVal*, float *maxVal*)
constrain a value to lie between two further values

Parameters:

[in, out] **v** vector

[in] **minVal** minimum value

[in] **maxVal** maximum value

void **glm_vec3_lerp** (vec3 *from*, vec3 *to*, float *t*, vec3 *dest*)
linear interpolation between two vector

formula: from + s * (to - from)

Parameters:

[in] **from** from value

[in] **to** to value

[in] **t** interpolant (amount) clamped between 0 and 1

[out] **dest** destination

5.13 vec3 extra

Header: cglm/vec3-ext.h

There are some functions are in called in extra header. These are called extra because they are not used like other functions in vec3.h in the future some of these functions ma be moved to vec3 header.

5.13.1 Table of contents (click to go):

Functions:

5.13.2 Functions documentation

void **glm_vec3_mulv**(vec3 *a*, vec3 *b*, vec3 *d*)

multiplies individual items

Parameters:

[in] **a** vec1

[in] **b** vec2

[out] **d** destination (v1[0] * v2[0], v1[1] * v2[1], v1[2] * v2[2])

void **glm_vec3_broadcast**(float *val*, vec3 *d*)

fill a vector with specified value

Parameters:

[in] **val** value

[out] **dest** destination

bool **glm_vec3_eq**(vec3 *v*, float *val*)

check if vector is equal to value (without epsilon)

Parameters:

[in] **v** vector

[in] **val** value

bool **glm_vec3_eq_eps**(vec3 *v*, float *val*)

check if vector is equal to value (with epsilon)

Parameters:

[in] **v** vector

[in] **val** value

bool **glm_vec3_eq_all**(vec3 *v*)
check if vectors members are equal (without epsilon)

Parameters:

[in] **v** vector

bool **glm_vec3_eqv**(vec3 *v1*, vec3 *v2*)
check if vector is equal to another (without epsilon) vector

Parameters:

[in] **vec** vector 1

[in] **vec** vector 2

bool **glm_vec3_eqv_eps**(vec3 *v1*, vec3 *v2*)
check if vector is equal to another (with epsilon)

Parameters:

[in] **v1** vector1

[in] **v2** vector2

float **glm_vec3_max**(vec3 *v*)
max value of vector

Parameters:

[in] **v** vector

float **glm_vec3_min**(vec3 *v*)

min value of vector

Parameters:

[in] **v** vector

bool **glm_vec3_isnan**(vec3 *v*)

check if one of items is NaN (not a number)
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **glm_vec3_isinf**(vec3 *v*)

check if one of items is INFINITY
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **glm_vec3_isvalid**(vec3 *v*)

check if all items are valid number

you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

void **glm_vec3_sign** (vec3 *v*, vec3 *dest*)

get sign of 32 bit float as +1, -1, 0

Parameters:

[in] **v** vector

[out] **dest** sign vector (only keeps signs as -1, 0, -1)

void **glm_vec3_sqrt** (vec3 *v*, vec3 *dest*)

square root of each vector item

Parameters:

[in] **v** vector

[out] **dest** destination vector (sqrt(v))

5.14 vec4

Header: cglm/vec4.h

5.14.1 Table of contents (click to go):

Macros:

1. glm_vec4_dup3(v, dest)
2. glm_vec4_dup(v, dest)
3. GLM_VEC4_ONE_INIT
4. GLM_VEC4_BLACK_INIT
5. GLM_VEC4_ZERO_INIT
6. GLM_VEC4_ONE
7. GLM_VEC4_BLACK
8. GLM_VEC4_ZERO

Functions:

1. `glm_vec4()`
2. `glm_vec4_copy3()`
3. `glm_vec4_copy()`
4. `glm_vec4_ucopy()`
5. `glm_vec4_zero()`
6. `glm_vec4_one()`
7. `glm_vec4_dot()`

8. `glm_vec4_norm2()`
9. `glm_vec4_norm()`
10. `glm_vec4_add()`
11. `glm_vec4_adds()`
12. `glm_vec4_sub()`
13. `glm_vec4_subs()`
14. `glm_vec4_mul()`
15. `glm_vec4_scale()`
16. `glm_vec4_scale_as()`
17. `glm_vec4_div()`
18. `glm_vec4_divs()`
19. `glm_vec4_addadd()`
20. `glm_vec4_subadd()`
21. `glm_vec4_muladd()`
22. `glm_vec4_muladds()`
23. `glm_vec4_maxadd()`
24. `glm_vec4_minadd()`
25. `glm_vec4_flipsign()`
26. `glm_vec4_flipsign_to()`
27. `glm_vec4_inv()`
28. `glm_vec4_inv_to()`
29. `glm_vec4_negate()`
30. `glm_vec4_negate_to()`
31. `glm_vec4_normalize()`
32. `glm_vec4_normalize_to()`
33. `glm_vec4_distance()`
34. `glm_vec4_maxv()`
35. `glm_vec4_minv()`
36. `glm_vec4_clamp()`
37. `glm_vec4_lerp()`
38. `glm_vec4_cubic()`

5.14.2 Functions documentation

void **glm_vec4** (vec3 *v3*, float *last*, vec4 *dest*)

init vec4 using vec3, since you are initializing vec4 with vec3 you need to set last item. cglm could set it zero but making it parameter gives more control

Parameters:

[in] **v3** vector4

[in] **last** last item of vec4

[out] **dest** destination

void **glm_vec4_copy3** (vec4 *a*, vec3 *dest*)

copy first 3 members of [a] to [dest]

Parameters:

[in] **a** source

[out] **dest** destination

void **glm_vec4_copy** (vec4 *v*, vec4 *dest*)

copy all members of [a] to [dest]

Parameters:

[in] **v** source

[in] **dest** destination

void **glm_vec4_ucopy** (vec4 *v*, vec4 *dest*)

copy all members of [a] to [dest]

alignment is not required

Parameters:

[in] **v** source

[in] **dest** destination

void **glm_vec4_zero** (vec4 *v*)

makes all members zero

Parameters:

[in, out] **v** vector

float **glm_vec4_dot** (vec4 *a*, vec4 *b*)

dot product of vec4

Parameters:

[in] **a** vector1

[in] **b** vector2

Returns: dot product

float **glm_vec4_norm2** (vec4 *v*)

norm * norm (magnitude) of vector

we can use this func instead of calling norm * norm, because it would call sqrtf fuction twice but with this func we can avoid func call, maybe this is not good name for this func

Parameters:

[in] **v** vector

Returns: square of norm / magnitude

float **glm_vec4_norm** (vec4 *vec*)

euclidean norm (magnitude), also called L2 norm

this will give magnitude of vector in euclidean space

Parameters:

[in] **vec** vector

void **glm_vec4_add**(vec4 *a*, vec4 *b*, vec4 *dest*)

add a vector to b vector store result in dest

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** destination vector

void **glm_vec4_adds**(vec4 *v*, float *s*, vec4 *dest*)

add scalar to v vector store result in dest (d = v + vec(s))

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec4_sub**(vec4 *a*, vec4 *b*, vec4 *dest*)

subtract b vector from a vector store result in dest (d = v1 - v2)

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** destination vector

void **glm_vec4_subs**(vec4 *v*, float *s*, vec4 *dest*)

subtract scalar from v vector store result in dest (d = v - vec(s))

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec4_mul**(vec4 *a*, vec4 *b*, vec4 *d*)

multiply two vector (component-wise multiplication)

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** result = (a[0] * b[0], a[1] * b[1], a[2] * b[2], a[3] * b[3])

void **glm_vec4_scale**(vec4 *v*, float *s*, vec4 *dest*)

multiply/scale vec4 vector with scalar: result = v * s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec4_scale_as**(vec4 *v*, float *s*, vec4 *dest*)
make vec4 vector scale as specified: result = unit(v) * s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec4_div**(vec4 *a*, vec4 *b*, vec4 *dest*)
div vector with another component-wise division: d = v1 / v2

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** result = (a[0] / b[0], a[1] / b[1], a[2] / b[2], a[3] / b[3])

void **glm_vec4_divs**(vec4 *v*, float *s*, vec4 *dest*)
div vector with scalar: d = v / s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** result = (a[0] / s, a[1] / s, a[2] / s, a[3] / s)

void **glm_vec4_addadd**(vec4 *a*, vec4 *b*, vec4 *dest*)

add two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a + b)

void **glm_vec4_subadd**(vec4 *a*, vec4 *b*, vec4 *dest*)

sub two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a - b)

void **glm_vec4_muladd**(vec4 *a*, vec4 *b*, vec4 *dest*)

mul two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a * b)

void **glm_vec4_muladds**(vec4 *a*, float *s*, vec4 *dest*)

mul vector with scalar and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector

[in] **s** scalar

[out] **dest** dest += (a * b)

void **glm_vec4_maxadd**(vec4 *a*, vec4 *b*, vec4 *dest*)

add max of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a * b)

void **glm_vec4_minadd**(vec4 *a*, vec4 *b*, vec4 *dest*)

add min of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** dest += (a * b)

void **glm_vec4_flipsign**(vec4 *v*)

DEPRACATED!

use `glm_vec4_negate()`

Parameters: | *[in, out]* **v** vector

void **glm_vec4_flipsign_to**(vec4 *v*, vec4 *dest*)

DEPRACATED!

use `glm_vec4_negate_to()`

Parameters:

[in] **v** vector

[out] **dest** negated vector

void **glm_vec4_inv** (vec4 *v*)

DEPRACATED!

use *glm_vec4_negate()*

Parameters:

[in, out] **v** vector

void **glm_vec4_inv_to** (vec4 *v*, vec4 *dest*)

DEPRACATED!

use *glm_vec4_negate_to()*

Parameters:

[in] **v** source

[out] **dest** destination

void **glm_vec4_negate** (vec4 *v*)

negate vector components

Parameters: | *[in, out]* **v** vector

void **glm_vec4_negate_to** (vec4 *v*, vec4 *dest*)

negate vector components and store result in dest

Parameters:

[in] **v** vector

[out] **dest** negated vector

void **glm_vec4_normalize** (vec4 *v*)

normalize vec4 and store result in same vec

Parameters:

[in, out] **v** vector

void **glm_vec4_normalize_to** (vec4 *vec*, vec4 *dest*)

normalize vec4 to dest

Parameters:

[in] **vec** source

[out] **dest** destination

float **glm_vec4_distance** (vec4 *v1*, vec4 *v2*)

distance between two vectors

Parameters:

[in] **mat** vector1

[in] **row1** vector2

Returns:

distance

void **glm_vec4_maxv** (vec4 *v1*, vec4 *v2*, vec4 *dest*)

max values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec4_minv** (vec4 *v1*, vec4 *v2*, vec4 *dest*)
min values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec4_clamp** (vec4 *v*, float *minVal*, float *maxVal*)
constrain a value to lie between two further values

Parameters:

[in, out] **v** vector

[in] **minVal** minimum value

[in] **maxVal** maximum value

void **glm_vec4_lerp** (vec4 *from*, vec4 *to*, float *t*, vec4 *dest*)
linear interpolation between two vector

formula: from + s * (to - from)

Parameters:

[in] **from** from value

[in] **to** to value

[in] **t** interpolant (amount) clamped between 0 and 1

[out] **dest** destination

void **glm_vec4_cubic** (float *s*, vec4 *dest*)
helper to fill vec4 as [S^3, S^2, S, 1]

Parameters:

[in] **s** parameter

[out] **dest** destination

5.15 vec4 extra

Header: cglm/vec4-ext.h

There are some functions are in called in extra header. These are called extra because they are not used like other functions in vec4.h in the future some of these functions ma be moved to vec4 header.

5.15.1 Table of contents (click to go):

Functions:

4. `glm_vec4_eq_eps()`
5. `glm_vec4_eq_all()`
6. `glm_vec4_eqv()`
7. `glm_vec4_eqv_eps()`
8. `glm_vec4_max()`
9. `glm_vec4_min()`

5.15.2 Functions documentation

void **glm_vec4_mulv**(vec4 *a*, vec4 *b*, vec4 *d*)
multiplies individual items

Parameters:

[in] **a** vec1
[in] **b** vec2
[out] **d** destination

void **glm_vec4_broadcast**(float *val*, vec4 *d*)
fill a vector with specified value

Parameters:

[in] **val** value
[out] **dest** destination

bool **glm_vec4_eq**(vec4 *v*, float *val*)
check if vector is equal to value (without epsilon)

Parameters:

[in] **v** vector
[in] **val** value

bool **glm_vec4_eq_eps**(vec4 *v*, float *val*)
check if vector is equal to value (with epsilon)

Parameters:

[in] **v** vector
[in] **val** value

bool **glm_vec4_eq_all**(vec4 *v*)
check if vectors members are equal (without epsilon)

Parameters:

[in] **v** vector

bool **glm_vec4_eqv**(vec4 *v1*, vec4 *v2*)
check if vector is equal to another (without epsilon) vector

Parameters:

[in] **vec** vector 1
[in] **vec** vector 2

bool **glm_vec4_eqv_eps**(vec4 *v1*, vec4 *v2*)
check if vector is equal to another (with epsilon)

Parameters:

[in] **v1** vector1

[in] **v2** vector2

float **glm_vec4_max**(vec4 *v*)
max value of vector

Parameters:

[in] **v** vector

float **glm_vec4_min**(vec4 *v*)

min value of vector

Parameters:

[in] **v** vector

bool **glm_vec4_isnan**(vec4 *v*)

check if one of items is NaN (not a number)
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **glm_vec4_isinf**(vec4 *v*)

check if one of items is INFINITY
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **glm_vec4_isvalid**(vec4 *v*)

check if all items are valid number
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

void **glm_vec4_sign**(vec4 *v*, vec4 *dest*)
get sign of 32 bit float as +1, -1, 0

Parameters:

[in] **v** vector

[out] **dest** sign vector (only keeps signs as -1, 0, -1)

void **glm_vec4_sqrt**(vec4 *v*, vec4 *dest*)
square root of each vector item

Parameters:

[in] **v** vector
[out] **dest** destination vector (sqrt(v))

5.16 vec2

Header: cglm/vec2.h

5.16.1 Table of contents (click to go):

Macros:

1. GLM_vec2_ONE_INIT
2. GLM_vec2_ZERO_INIT
3. GLM_vec2_ONE
4. GLM_vec2_ZERO

Functions:

1. *glm_vec2()*
2. *glm_vec2_copy()*
3. *glm_vec2_zero()*
4. *glm_vec2_one()*
5. *glm_vec2_dot()*
6. *glm_vec2_cross()*
7. *glm_vec2_norm2()*
8. *glm_vec2_norm()*
9. *glm_vec2_add()*
10. *glm_vec2_adds()*
11. *glm_vec2_sub()*
12. *glm_vec2_subs()*
13. *glm_vec2_mul()*
14. *glm_vec2_scale()*
15. *glm_vec2_scale_as()*
16. *glm_vec2_div()*
17. *glm_vec2_divs()*
18. *glm_vec2_addadd()*
19. *glm_vec2_subadd()*
20. *glm_vec2_muladd()*
21. *glm_vec2_muladds()*
22. *glm_vec2_maxadd()*

5.16.2 Functions documentation

void **glm_vec2** (float * *v*, vec2 *dest*)
init vec2 using vec3 or vec4

Parameters:

[in] **v** vector
[out] **dest** destination

void **glm_vec2_copy** (vec2 *a*, vec2 *dest*)
copy all members of [a] to [dest]

Parameters:

[in] **a** source
[out] **dest** destination

void **glm_vec2_zero** (vec2 *v*)
makes all members 0.0f (zero)

Parameters:

[in, out] **v** vector

void **glm_vec2_one** (vec2 *v*)
makes all members 1.0f (one)

Parameters:

[in, out] **v** vector

float **glm_vec2_dot** (vec2 *a*, vec2 *b*)
dot product of vec2

Parameters:

[in] **a** vector1
[in] **b** vector2

Returns: dot product

void **glm_vec2_cross** (vec2 *a*, vec2 *b*, vec2 *d*)
cross product of two vector (RH)

ref: http://allenchou.net/2013/07/cross-product-of-2d-vectors/

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** destination

Returns: Z component of cross product

float **glm_vec2_norm2** (vec2 *v*)
norm * norm (magnitude) of vector

we can use this func instead of calling norm * norm, because it would call sqrtf fuction twice but with this func we can avoid func call, maybe this is not good name for this func

Parameters:

[in] **v** vector

Returns: square of norm / magnitude

float **glm_vec2_norm** (vec2 *vec*)

euclidean norm (magnitude), also called L2 norm
this will give magnitude of vector in euclidean space

Parameters:

[in] **vec** vector

void **glm_vec2_add** (vec2 *a*, vec2 *b*, vec2 *dest*)
add a vector to b vector store result in dest

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** destination vector

void **glm_vec2_adds** (vec2 *a*, float *s*, vec2 *dest*)
add scalar to v vector store result in dest (d = v + vec(s))

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec2_sub** (vec2 *v1*, vec2 *v2*, vec2 *dest*)
subtract b vector from a vector store result in dest (d = v1 - v2)

Parameters:

[in] **a** vector1

[in] **b** vector2

[out] **dest** destination vector

void **glm_vec2_subs** (vec2 *v*, float *s*, vec2 *dest*)
subtract scalar from v vector store result in dest (d = v - vec(s))

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec2_mul** (vec2 *a*, vec2 *b*, vec2 *d*)
multiply two vector (component-wise multiplication)

Parameters:

[in] **a** vector

[in] **b** scalar

[out] **d** result = (a[0] * b[0], a[1] * b[1], a[2] * b[2])

void **glm_vec2_scale** (vec2 *v*, float *s*, vec2 *dest*)

multiply/scale vec2 vector with scalar: result = v * s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec2_scale_as** (vec2 *v*, float *s*, vec2 *dest*)
make vec2 vector scale as specified: result = unit(v) * s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** destination vector

void **glm_vec2_div** (vec2 *a*, vec2 *b*, vec2 *dest*)
div vector with another component-wise division: d = a / b

Parameters:

[in] **a** vector 1

[in] **b** vector 2

[out] **dest** result = (a[0] / b[0], a[1] / b[1], a[2] / b[2])

void **glm_vec2_divs** (vec2 *v*, float *s*, vec2 *dest*)
div vector with scalar: d = v / s

Parameters:

[in] **v** vector

[in] **s** scalar

[out] **dest** result = (a[0] / s, a[1] / s, a[2] / s])

void **glm_vec2_addadd** (vec2 *a*, vec2 *b*, vec2 *dest*)

add two vectors and add result to sum

it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a + b)

void **glm_vec2_subadd**(vec2 *a*, vec2 *b*, vec2 *dest*)

sub two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a - b)

void **glm_vec2_muladd**(vec2 *a*, vec2 *b*, vec2 *dest*)

mul two vectors and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a * b)

void **glm_vec2_muladds**(vec2 *a*, float *s*, vec2 *dest*)

mul vector with scalar and add result to sum
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector
[in] **s** scalar
[out] **dest** dest += (a * b)

void **glm_vec2_maxadd**(vec2 *a*, vec2 *b*, vec2 *dest*)

add max of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a * b)

void **glm_vec2_minadd**(vec2 *a*, vec2 *b*, vec2 *dest*)

add min of two vector to result/dest
it applies += operator so dest must be initialized

Parameters:

[in] **a** vector 1
[in] **b** vector 2
[out] **dest** dest += (a * b)

void **glm_vec2_negate**(vec2 *v*)
negate vector components

Parameters:

[in, out] **v** vector

void **glm_vec2_negate_to**(vec2 *v*, vec2 *dest*)
negate vector components and store result in dest

Parameters:

[in] **v** vector
[out] **dest** negated vector

void **glm_vec2_normalize**(vec2 *v*)
normalize vec2 and store result in same vec

Parameters:

[in, out] **v** vector

void **glm_vec2_normalize_to**(vec2 *vec*, vec2 *dest*)

normalize vec2 to dest

Parameters:

[in] **vec** source
[out] **dest** destination

void **glm_vec2_rotate**(vec2 *v*, float *angle*, vec2 *dest*)

rotate vec2 around axis by angle using Rodrigues' rotation formula

Parameters:

[in] **v** vector
[in] **axis** axis vector
[out] **dest** destination

float **glm_vec2_distance2**(vec2 *v1*, vec2 *v2*)
squared distance between two vectors

Parameters:

[in] **mat** vector1
[in] **row1** vector2

Returns:

squared distance (distance * distance)

float **glm_vec2_distance**(vec2 *v1*, vec2 *v2*)
distance between two vectors

Parameters:

[in] **mat** vector1

[in] **row1** vector2

Returns:

distance

void **glm_vec2_maxv**(vec2 *v1*, vec2 *v2*, vec2 *dest*)
max values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec2_minv**(vec2 *v1*, vec2 *v2*, vec2 *dest*)
min values of vectors

Parameters:

[in] **v1** vector1

[in] **v2** vector2

[out] **dest** destination

void **glm_vec2_clamp**(vec2 *v*, float *minVal*, float *maxVal*)
constrain a value to lie between two further values

Parameters:

[in, out] **v** vector

[in] **minVal** minimum value

[in] **maxVal** maximum value

void **glm_vec2_lerp**(vec2 *from*, vec2 *to*, float *t*, vec2 *dest*)
linear interpolation between two vector

formula: from + s * (to - from)

Parameters:

[in] **from** from value

[in] **to** to value

[in] **t** interpolant (amount) clamped between 0 and 1

[out] **dest** destination

5.17 vec2 extra

Header: cglm/vec2-ext.h

There are some functions are in called in extra header. These are called extra because they are not used like other functions in vec2.h in the future some of these functions ma be moved to vec2 header.

5.17.1 Table of contents (click to go):

Functions:

5.17.2 Functions documentation

void **glm_vec2_fill** (vec2 *v*, float *val*)
fill a vector with specified value

Parameters:

[in,out] **dest** destination

[in] **val** value

bool **glm_vec2_eq** (vec2 *v*, float *val*)
check if vector is equal to value (without epsilon)

Parameters:

[in] **v** vector

[in] **val** value

bool **glm_vec2_eq_eps** (vec2 *v*, float *val*)
check if vector is equal to value (with epsilon)

Parameters:

[in] **v** vector

[in] **val** value

bool **glm_vec2_eq_all** (vec2 *v*)
check if vectors members are equal (without epsilon)

Parameters:

[in] **v** vector

bool **`glm_vec2_eqv`** (vec2 *v1*, vec2 *v2*)
check if vector is equal to another (without epsilon) vector

Parameters:

[in] **vec** vector 1

[in] **vec** vector 2

bool **`glm_vec2_eqv_eps`** (vec2 *v1*, vec2 *v2*)
check if vector is equal to another (with epsilon)

Parameters:

[in] **v1** vector1

[in] **v2** vector2

float **`glm_vec2_max`** (vec2 *v*)
max value of vector

Parameters:

[in] **v** vector

float **`glm_vec2_min`** (vec2 *v*)

min value of vector

Parameters:

[in] **v** vector

bool **`glm_vec2_isnan`** (vec2 *v*)

check if one of items is NaN (not a number)
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **`glm_vec2_isinf`** (vec2 *v*)

check if one of items is INFINITY
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

bool **`glm_vec2_isvalid`** (vec2 *v*)

check if all items are valid number
you should only use this in DEBUG mode or very critical asserts

Parameters:

[in] **v** vector

void **glm_vec2_sign** (vec2 *v*, vec2 *dest*)
get sign of 32 bit float as +1, -1, 0

Parameters:

[in] **v** vector

[out] **dest** sign vector (only keeps signs as -1, 0, -1)

void **glm_vec2_sqrt** (vec2 *v*, vec2 *dest*)
square root of each vector item

Parameters:

[in] **v** vector

[out] **dest** destination vector (sqrt(v))

5.18 color

Header: cglm/color.h

5.18.1 Table of contents (click to go):

Functions:

1. *glm_luminance()*

5.18.2 Functions documentation

float **glm_luminance** (vec3 *rgb*)

averages the color channels into one value

This function uses formula in COLLADA 1.5 spec which is

```
luminance = (color.r * 0.212671) +
            (color.g * 0.715160) +
            (color.b * 0.072169)
```

It is based on the ISO/CIE color standards (see ITU-R Recommendation BT.709-4), that averages the color channels into one value

Parameters:

[in] **rgb** RGB color

5.19 plane

Header: cglm/plane.h

Plane extract functions are in frustum header and documented in *frustum* page.

Definition of plane:

Plane equation: **Ax + By + Cz + D = 0**

Plan is stored in **vec4** as **[A, B, C, D]**. (A, B, C) is normal and D is distance

5.19.1 Table of contents (click to go):

Functions:

5.19.2 Functions documentation

void **glm_plane_normalize**(vec4 *plane*)

normalizes a plane

Parameters:

[in, out] **plane** pnale to normalize

5.20 Project / UnProject

Header: cglm/project.h

Viewport is required as *vec4* **[X, Y, Width, Height]** but this doesn't mean that you should store it as **vec4**. You can convert your data representation to vec4 before passing it to related functions.

5.20.1 Table of contents (click to go):

Functions:

5.20.2 Functions documentation

void **glm_unprojecti**(vec3 *pos*, mat4 *invMat*, vec4 *vp*, vec3 *dest*)

maps the specified viewport coordinates into specified space [1] the matrix should contain projection matrix.

if you don't have (and don't want to have) an inverse matrix then use glm_unproject version. You may use existing inverse of matrix in somewhere else, this is why glm_unprojecti exists to save save inversion cost

[1] space:

- if m = invProj: View Space
- if m = invViewProj: World Space
- if m = invMVP: Object Space

You probably want to map the coordinates into object space so use invMVP as m

Computing viewProj:

```
glm_mat4_mul(proj, view, viewProj);
glm_mat4_mul(viewProj, model, MVP);
glm_mat4_inv(viewProj, invMVP);
```

Parameters:

[in] **pos** point/position in viewport coordinates
[in] **invMat** matrix (see brief)
[in] **vp** viewport as [x, y, width, height]
[out] **dest** unprojected coordinates

void **glm_unproject** (vec3 *pos*, mat4 *m*, vec4 *vp*, vec3 *dest*)

maps the specified viewport coordinates into specified space [1] the matrix should contain projection matrix.

this is same as glm_unprojecti except this function get inverse matrix for you.

[1] space:

- if m = proj: View Space
- if m = viewProj: World Space
- if m = MVP: Object Space

You probably want to map the coordinates into object space so use MVP as m

Computing viewProj and MVP:

```
glm_mat4_mul(proj, view, viewProj);
glm_mat4_mul(viewProj, model, MVP);
```

Parameters:

[in] **pos** point/position in viewport coordinates
[in] **m** matrix (see brief)
[in] **vp** viewport as [x, y, width, height]
[out] **dest** unprojected coordinates

void **glm_project** (vec3 *pos*, mat4 *m*, vec4 *vp*, vec3 *dest*)

map object coordinates to window coordinates

Computing MVP:

```
glm_mat4_mul(proj, view, viewProj);
glm_mat4_mul(viewProj, model, MVP);
```

this could be useful for gettng a bbox which fits with view frustum and object bounding boxes. In this case you crop view frustum box with objects box

Parameters:

[in] **pos** object coordinates
[in] **m** MVP matrix

[in] **vp** viewport as [x, y, width, height]

[out] **dest** projected coordinates

5.21 utils / helpers

Header: cglm/util.h

5.21.1 Table of contents (click to go):

Functions:

5.21.2 Functions documentation

int **glm_sign**(int *val*)

returns sign of 32 bit integer as +1, -1, 0

Important: It returns 0 for zero input

Parameters:

[in] **val** an integer

Returns: sign of given number

float **glm_signf**(float *val*)

returns sign of 32 bit integer as +1.0, -1.0, 0.0

Important: It returns 0.0f for zero input

Parameters:

[in] **val** a float

Returns: sign of given number

float **glm_rad**(float *deg*)

convert degree to radians

Parameters:

[in] **deg** angle in degrees

float **glm_deg**(float *rad*)

convert radians to degree

Parameters:

[in] **rad** angle in radians

void **glm_make_rad**(float **degm*)

convert exsisting degree to radians. this will override degrees value

Parameters:

[in, out] **deg** pointer to angle in degrees

void **glm_make_deg**(float **rad*)

convert exsisting radians to degree. this will override radians value

Parameters:

[in, out] **rad** pointer to angle in radians

float **glm_pow2**(float *x*)

multiplies given parameter with itself = x * x or powf(x, 2)

Parameters:

[in] **x** value

Returns: square of a given number

float **glm_min**(float *a*, float *b*)

returns minimum of given two values

Parameters:

[in] **a** number 1

[in] **b** number 2

Returns: minimum value

float **glm_max** (float *a*, float *b*)

returns maximum of given two values

Parameters:

[in] **a** number 1

[in] **b** number 2

Returns: maximum value

void **glm_clamp** (float *val*, float *minVal*, float *maxVal*)

constrain a value to lie between two further values

Parameters:

[in] **val** input value

[in] **minVal** minimum value

[in] **maxVal** maximum value

Returns: clamped value

float **glm_lerp** (float *from*, float *to*, float *t*)

linear interpolation between two number

formula: from + s * (to - from)

Parameters:

[in] **from** from value

[in] **to** to value

[in] **t** interpolant (amount) clamped between 0 and 1

Returns: interpolated value

bool **glm_eq** (float *a*, float *b*)

check if two float equal with using EPSILON

Parameters:

[in] **a** a

[in] **b** b

Returns: true if a and b are equal

float **glm_percent** (float *from*, float *to*, float *current*)

percentage of current value between start and end value

Parameters:

[in] **from** from value

[in] **to** to value

[in] **current** value between from and to values

Returns: percentage of current value

float **glm_percentc** (float *from*, float *to*, float *current*)
clamped percentage of current value between start and end value

Parameters:

[in] **from** from value

[in] **to** to value

[in] **current** value between from and to values

Returns: clamped normalized percent (0-100 in 0-1)

void **glm_swapf** (float **a*, float **b*)
swap two float values

Parameters:

[in] **a** float 1

[in] **b** float 2

5.22 io (input / output e.g. print)

Header: cglm/io.h

There are some built-in print functions which may save your time, especially for debugging.

All functions accept **FILE** parameter which makes very flexible. You can even print it to file on disk.

In general you will want to print them to console to see results. You can use **stdout** and **stderr** to write results to console. Some programs may occupy **stdout** but you can still use **stderr**. Using **stderr** is suggested.

Example to print mat4 matrix:

```
mat4 transform;
/* ... */
glm_mat4_print(transform, stderr);
```

NOTE: print functions use **%0.4f** precision if you need more (you probably will in some cases), you can change it temporary. cglm may provide precision parameter in the future

Changes since **v0.7.3**: * Now mis-alignment of columns are fixed: larger numbers are printed via %g and others are printed via %f. Column withs are calculated before print. * Now values are colorful ;) * Some print improvements * New options with default values:

```
#define CGLM_PRINT_PRECISION    5
#define CGLM_PRINT_MAX_TO_SHORT 1e5
#define CGLM_PRINT_COLOR        "\033[36m"
#define CGLM_PRINT_COLOR_RESET  "\033[0m"
```

- Inline prints are only enabled in DEBUG mode and if **CGLM_DEFINE_PRINTS** is defined.

Check options page.

5.22.1 Table of contents (click to go):

Functions:

1. `glm_mat4_print()`
2. `glm_mat3_print()`

3. *glm_vec4_print()*
4. *glm_vec3_print()*
5. *glm_ivec3_print()*
6. *glm_versor_print()*
7. *glm_aabb_print()*

5.22.2 Functions documentation

void **glm_mat4_print**(mat4 *matrix*, FILE * __restrict *ostream*)

print mat4 to given stream

Parameters:

[in] **matrix** matrix
[in] **ostream** FILE to write

void **glm_mat3_print**(mat3 *matrix*, FILE * __restrict *ostream*)

print mat3 to given stream

Parameters:

[in] **matrix** matrix
[in] **ostream** FILE to write

void **glm_vec4_print**(vec4 *vec*, FILE * __restrict *ostream*)

print vec4 to given stream

Parameters:

[in] **vec** vector
[in] **ostream** FILE to write

void **glm_vec3_print**(vec3 *vec*, FILE * __restrict *ostream*)

print vec3 to given stream

Parameters:

[in] **vec** vector
[in] **ostream** FILE to write

void **glm_ivec3_print**(ivec3 *vec*, FILE * __restrict *ostream*)

print ivec3 to given stream

Parameters:

[in] **vec** vector
[in] **ostream** FILE to write

void **glm_versor_print** (versor *vec*, FILE * __restrict *ostream*)

print quaternion to given stream

Parameters:

[in] **vec** quaternion
[in] **ostream** FILE to write

void **glm_aabb_print** (versor *vec*, const char * __restrict *tag*, FILE * __restrict *ostream*)

print aabb to given stream

Parameters:

[in] **vec** aabb (axis-aligned bounding box)
[in] **tag** tag to find it more easly in logs
[in] **ostream** FILE to write

5.23 precompiled functions (call)

All funcitons in **glm_** namespace are forced to **inline**. Most functions also have pre-compiled version.

Precompiled versions are in **glmc_** namespace. *c* in the namespace stands for "call".

Since precompiled functions are just wrapper for inline verisons, these functions are not documented individually. It would be duplicate documentation also it would be hard to sync documentation between inline and call verison for me.

By including **clgm/cglm.h** you include all inline verisons. To get precompiled versions you need to include **cglm/call.h** header it also includes all call versions plus *clgm/cglm.h* (inline verisons)

5.24 Sphere

Header: cglm/sphere.h

Definition of sphere:

Sphere Representation in cglm is *vec4*: **[center.x, center.y, center.z, radii]**

You can call any vec3 function by pasing sphere. Because first three elements defines center of sphere.

5.24.1 Table of contents (click to go):

Functions:

1. `glm_sphere_radii()`
2. `glm_sphere_transform()`
3. `glm_sphere_merge()`

4. *glm_sphere_sphere()*
5. *glm_sphere_point()*

5.24.2 Functions documentation

float **glm_sphere_radii**(vec4 *s*)

helper for getting sphere radius

Parameters:

[in] **s** sphere

Returns: returns radii

void **glm_sphere_transform**(vec4 *s*, mat4 *m*, vec4 *dest*)

apply transform to sphere, it is just wrapper for glm_mat4_mulv3

Parameters:

[in] **s** sphere
[in] **m** transform matrix
[out] **dest** transformed sphere

void **glm_sphere_merge**(vec4 *s1*, vec4 *s2*, vec4 *dest*)

merges two spheres and creates a new one

two sphere must be in same space, for instance if one in world space then the other must be in world space too, not in local space.

Parameters:

[in] **s1** sphere 1
[in] **s2** sphere 2
[out] **dest** merged/extended sphere

bool **glm_sphere_sphere**(vec4 *s1*, vec4 *s2*)

check if two sphere intersects

Parameters:

[in] **s1** sphere
[in] **s2** other sphere

bool **glm_sphere_point**(vec4 *s*, vec3 *point*)

check if sphere intersects with point

Parameters:

[in] **s** sphere

[in] **point** point

5.25 Curve

Header: cglm/curve.h

Common helpers for common curves. For specific curve see its header/doc e.g bezier

5.25.1 Table of contents (click to go):

Functions:

5.25.2 Functions documentation

float **glm_smc**(float *s*, mat4 *m*, vec4 *c*)

helper function to calculate **S** * **M** * **C** multiplication for curves

this function does not encourage you to use SMC, instead it is a helper if you use SMC.

if you want to specify S as vector then use more generic glm_mat4_rmc() func.

Example usage:

```
Bs = glm_smc(s, GLM_BEZIER_MAT, (vec4){p0, c0, c1, p1})
```

Parameters:

[in] **s** parameter between 0 and 1 (this will be [s3, s2, s, 1])
[in] **m** basis matrix
[out] **c** position/control vector

Returns: scalar value e.g. Bs

5.26 Bezier

Header: cglm/bezier.h

Common helpers for cubic bezier and similar curves.

5.26.1 Table of contents (click to go):

Functions:

5.26.2 Functions documentation

float **glm_bezier** (float *s*, float *p0*, float *c0*, float *c1*, float *p1*)

cubic bezier interpolation
formula:

```
B(s) = P0*(1-s)^3 + 3*C0*s*(1-s)^2 + 3*C1*s^2*(1-s) + P1*s^3
```

similar result using matrix:

```
B(s) = glm_smc(t, GLM_BEZIER_MAT, (vec4){p0, c0, c1, p1})
```

glm_eq(glm_smc(. . .), glm_bezier(. . .)) should return TRUE

Parameters:

- *[in]* **s** parameter between 0 and 1
- *[in]* **p0** begin point
- *[in]* **c0** control point 1
- *[in]* **c1** control point 2
- *[in]* **p1** end point

Returns: B(s)

float **glm_hermite** (float *s*, float *p0*, float *t0*, float *t1*, float *p1*)

cubic hermite interpolation
formula:

```
H(s) = P0*(2*s^3 - 3*s^2 + 1) + T0*(s^3 - 2*s^2 + s) + P1*(-2*s^3 + 3*s^2) +
↪T1*(s^3 - s^2)
```

similar result using matrix:

```
H(s) = glm_smc(t, GLM_HERMITE_MAT, (vec4){p0, p1, c0, c1})
```

glm_eq(glm_smc(. . .), glm_hermite(. . .)) should return TRUE

Parameters:

[in] **s** parameter between 0 and 1
[in] **p0** begin point
[in] **t0** tangent 1
[in] **t1** tangent 2
[in] **p1** end point

Returns: B(s)

float **glm_decasteljau** (float *prm*, float *p0*, float *c0*, float *c1*, float *p1*)

iterative way to solve cubic equation

Parameters:

[in] **prm** parameter between 0 and 1
[in] **p0** begin point
[in] **c0** control point 1
[in] **c1** control point 2
[in] **p1** end point

Returns: parameter to use in cubic equation

5.27 version

Header: cglm/version.h

cglm uses semantic versioning (http://semver.org) which is MAJOR.MINOR.PATCH

CGLM_VERSION_MAJOR is major number of the version.
CGLM_VERSION_MINOR is minor number of the version.
CGLM_VERSION_PATCH is patch number of the version.

every release increases these numbers. You can check existing version by including *cglm/version.h*

5.28 ray

Header: cglm/ray.h

This is for collision-checks used by ray-tracers and the like.

5.28.1 Table of contents (click to go):

Functions:

5.28.2 Functions documentation

bool **glm_ray_triangle**(vec3 *origin*, vec3 *direction*, vec3 *v0*, vec3 *v1*, vec3 *v2*, float **d*)

Möller–Trumbore ray-triangle intersection algorithm

Parameters:

- *[in]* **origin** origin of ray
- *[in]* **direction** direction of ray
- *[in]* **v0** first vertex of triangle
- *[in]* **v1** second vertex of triangle
- *[in]* **v2** third vertex of triangle
- *[in, out]* **d** float pointer to save distance to intersection
- *[out]* **intersection** whether there is intersection

CHAPTER 6

Options

A few options are provided via macros.

6.1 Alignment Option

As default, cglm requires types to be aligned. Alignment requirements:

vec3: 8 byte vec4: 16 byte mat4: 16 byte versor: 16 byte

By starting **v0.4.5** cglm provides an option to disable alignment requirement. To enable this option define **CGLM_ALL_UNALIGNED** macro before all headers. You can define it in Xcode, Visual Studio (or other IDEs) or you can also prefer to define it in build system. If you use pre-compiled versions then you have to compile cglm with **CGLM_ALL_UNALIGNED** macro.

VERY VERY IMPORTANT: If you use cglm in multiple projects and those projects are depends on each other, then

> *ALWAYS* or *NEVER USE* **CGLM_ALL_UNALIGNED** macro in linked projects

> if you do not know what you are doing. Because a cglm header included via 'project A' may force types to be aligned and another cglm header included via 'project B' may not require alignment. In this case cglm functions will read from and write to **INVALID MEMORY LOCATIONs**.
>
> ALWAYS USE SAME CONFIGURATION / OPTION for **cglm** if you have multiple projects.
>
> For instance if you set CGLM_ALL_UNALIGNED in a project then set it in other projects too

6.2 Clipspace Option[s]

By starting **v0.8.3** cglm provides options to switch between clipspace configurations.

Clipspace related files are located at *include/cglm/[struct]/clipspace.h* but these are included in related files like *cam.h*. If you don't want to change your existing clipspace configuration and want to use different clipspace function like *glm_lookat_zo* or *glm_lookat_lh_zo*… then you can include individual headers or just define *CGLM_CLIPSPACE_INCLUDE_ALL* which will iclude all headers for you.

1. **CGLM_CLIPSPACE_INCLUDE_ALL**
2. **CGLM_FORCE_DEPTH_ZERO_TO_ONE**
3. **CGLM_FORCE_LEFT_HANDED**

1. **CGLM_CLIPSPACE_INCLUDE_ALL**:

By defining this macro, **cglm** will include all clipspace functions for you by just using *#include cglm/cglm.h* or *#include cglm/struct.h* or *#include cglm/call.h*

Otherwise you need to include header you want manually e.g. *#include cglm/clipspace/view_rh_zo.h*

2. **CGLM_FORCE_DEPTH_ZERO_TO_ONE**

This is similar to **GLM**'s **GLM_FORCE_DEPTH_ZERO_TO_ONE** option. This will set clip space between 0 to 1 which makes **cglm** Vulkan, Metal friendly.

You can use functions like *glm_lookat_lh_zo()* individually. By setting **CGLM_FORCE_DEPTH_ZERO_TO_ONE** functions in cam.h for instance will use *_zo* versions.

3. **CGLM_FORCE_LEFT_HANDED**

Force **cglm** to use the left handed coordinate system by default, currently **cglm** uses right handed coordinate system as default, you can change this behavior with this option.

VERY VERY IMPORTANT:

Be careful if you include **cglm** in multiple projects.

6.3 SSE and SSE2 Shuffle Option

_mm_shuffle_ps generates **shufps** instruction even if registers are same. You can force it to generate **pshufd** instruction by defining **CGLM_USE_INT_DOMAIN** macro. As default it is not defined.

6.4 SSE3 and SSE4 Dot Product Options

You have to extra options for dot product: **CGLM_SSE4_DOT** and **CGLM_SSE3_DOT**.

- If **SSE4** is enabled then you can define **CGLM_SSE4_DOT** to force cglm to use **_mm_dp_ps** instruction.
- If **SSE3** is enabled then you can define **CGLM_SSE3_DOT** to force cglm to use **_mm_hadd_ps** instructions.

otherwise cglm will use custom cglm's hadd functions which are optimized too.

6.5 Print Options

1. **CGLM_DEFINE_PRINTS**
2. **CGLM_NO_PRINTS_NOOP** (use CGLM_DEFINE_PRINTS)

Inline prints are only enabled in **DEBUG** mode or if **CGLM_DEFINE_PRINTS** is defined. **glmc_** versions will always print too.

Because **cglm** tried to enable print functions in debug mode and disable them in release/production mode to eliminate printing costs when we do not need them.

cglm checks **DEBUG** or **_DEBUG** macros to test debug mode, if these are not working for you then you can use **CGLM_DEFINE_PRINTS** to force enable, or create a PR to introduce new macro to test against debugging mode.

If DEBUG mode is not enabled then print functions will be emptied to eliminate print function calls. You can disable this feature too by defining **CGLM_DEFINE_PRINTS** macro top of cglm header or in project/build settings…

3. **CGLM_PRINT_PRECISION** 5

precision.

4. **CGLM_PRINT_MAX_TO_SHORT** 1e5

if a number is greater than this value then %g will be used, since this is shorten print you won't be able to see high precision.

5. **CGLM_PRINT_COLOR** "033[36m"
6. **CGLM_PRINT_COLOR_RESET** "033[0m"

You can disable colorful print output by defining **CGLM_PRINT_COLOR** and **CGLM_PRINT_COLOR_RESET** as empty macro. Because some terminals may not support colors.

CHAPTER 7

Troubleshooting

It is possible that sometimes you may get crashes or wrong results. Follow these topics

7.1 Memory Allocation:

Again, **cglm** doesn't alloc any memory on heap. cglm functions works like memcpy; it copies data from src, makes calculations then copy the result to dest.

You are responsible for allocation of **src** and **dest** parameters.

7.2 Alignment:

vec4 and **mat4** types requires 16 byte alignment. These types are marked with align attribute to let compiler know about this requirement.

But since MSVC (Windows) throws the error:

"formal parameter with requested alignment of 16 won't be aligned"

The alignment attribute has been commented for MSVC

```
#if defined(_MSC_VER)
#  define CGLM_ALIGN(X) /* __declspec(align(X)) */
#else
#  define CGLM_ALIGN(X) __attribute((aligned(X)))
#endif.
```

So MSVC may not know about alignment requirements when creating variables. The interesting thing is that, if I remember correctly Visual Studio 2017 doesn't throw the above error. So we may uncomment that line for Visual Studio 2017, you may do it yourself.

This MSVC issue is still in TODOs.

UPDATE: By starting v0.4.5 cglm provides an option to disable alignment requirement. Also alignment is disabled for older msvc verisons as default. Now alignment is only required in Visual Studio 2017 version 15.6+ if CGLM_ALL_UNALIGNED macro is not defined.

7.3 Crashes, Invalid Memory Access:

Probably you are trying to write to invalid memory location.

You may used wrong function for what you want to do.

For instance you may called **glm_vec4_** functions for **vec3** data type. It will try to write 32 byte but since **vec3** is 24 byte it should throw memory access error or exit the app without saying anything.

UPDATE - IMPORTANT:

> On MSVC or some other compilers, if alignment is enabled (default) then double check alignment requirements if you got a crash.
>
> If you send GLM_VEC4_ONE or similar macros directly to a function, it may be crashed.
>
> Because compiler may not apply alignment as defined on **typedef** to that macro while passing it (on stack) to a function.

7.4 Wrong Results:

Again, you may used wrong function.

For instance if you use **glm_normalize**() or **glm_vec3_normalize**() for **vec4**, it will assume that passed param is **vec3** and will normalize it for **vec3**. Since you need to **vec4** to be normalized in your case, you will get wrong results.

Accessing vec4 type with vec3 functions is valid, you will not get any error, exception or crash. You only get wrong results if you don't know what you are doing!

So be carefull, when your IDE (Xcode, Visual Studio ...) tried to autocomplete function names, READ IT :)

Also implementation may be wrong please let us know by creating an issue on Github.

7.5 BAD_ACCESS : Thread 1: EXC_BAD_ACCESS (code=EXC_I386_GPFLT) or Similar Errors/Crashes

This is similar issue with alignment. For instance if you compiled **cglm** with AVX (**-mavx**, intentionally or not) and if you use **cglm** in an environment that doesn't support AVX (or if AVX is disabled intentionally) e.g. environment that max support SSE2/3/4, then you probably get **BAD ACCESS** or similar...

Because if you compile **cglm** with AVX it aligns **mat4** with 32 byte boundary, and your project aligns that as 16 byte boundary...

Check alignment, supported vector extension or simd in **cglm** and linked projects...

7.6 Other Issues?

Please let us know by creating an issue on Github.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

G