**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Департамент программной инженерии

|  |  |  |
| --- | --- | --- |
| СОГЛАСОВАНО  Профессор  департамента программной инженерии  кандидат технических наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.М. Гринкруг  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. |  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Шилов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** | **RU.17701729.03.05-01** | | **БИБЛИОТЕКА ИНВЕРСНОЙ КИНЕМАТИКИ В WEB-ПРИЛОЖЕНИЯХ**  **Текст программы**  **ЛИСТ УТВЕРЖДЕНИЯ**  **RU.17701729.03.05-01 12 01-1-ЛУ** | | |
|  |  | |
| Исполнитель:  студентка группы БПИ162  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Казанцева А.Р. /  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | |
|  | | |
|  | |  |

**2019**

|  |  |  |  |
| --- | --- | --- | --- |
| **УТВЕРЖДЕНО**  **RU.17701729.03.05-01 12 01-1-ЛУ** | |  |  |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** | **RU.17701729.03.05-01** | | **БИБЛИОТЕКА ИНВЕРСНОЙ КИНЕМАТИКИ В WEB-ПРИЛОЖЕНИЯХ**  **Текст программы**  **RU.17701729.03.05-01 12 01-1**    **Листов 117** | | | | |
|  |  | | | |
|  | | | |
|  | | | | |

**2019**

СОДЕРЖАНИЕ

[АННОТАЦИЯ 4](#_Toc6871551)

[1. ТЕКСТ ПРОГРАММЫ 5](#_Toc6871552)

[1.1. Аналитический модуль 5](#_Toc6871553)

[1.1.1. BaseboneConstraintType2D 5](#_Toc6871554)

[1.1.2. BaseboneConstraintType3D 5](#_Toc6871555)

[1.1.3. BoneConnectionPoint 6](#_Toc6871556)

[1.1.4. ConstraintCoordinateSystem 6](#_Toc6871557)

[1.1.5. FabrikBone2D 6](#_Toc6871558)

[1.1.6. FabrikBone3D 12](#_Toc6871559)

[1.1.7. FabrikChain2D 19](#_Toc6871560)

[1.1.8. FabrikChain3D 31](#_Toc6871561)

[1.1.9. FabrikJoint2D 51](#_Toc6871562)

[1.1.10. FabrikJoint3D 54](#_Toc6871563)

[1.1.11. FabrikStructure2D 60](#_Toc6871564)

[1.1.12. FabrikStructure3D 63](#_Toc6871565)

[1.1.13. JointType 67](#_Toc6871566)

[1.2. Модуль визуализации 67](#_Toc6871567)

[1.2.1. Camera 67](#_Toc6871568)

[1.2.2. CameraController 68](#_Toc6871569)

[1.2.3. Modal 69](#_Toc6871570)

[1.2.4. Primatives.Primitive 70](#_Toc6871571)

[1.2.5. Primatives.Point 71](#_Toc6871572)

[1.2.6. Primatives.GridAxis 71](#_Toc6871573)

[1.2.7. RenderLoop 73](#_Toc6871574)

[1.2.8. Scene 74](#_Toc6871575)

[1.2.9. Point 77](#_Toc6871576)

[1.2.10. Primitive 77](#_Toc6871577)

[1.2.11. Shader 78](#_Toc6871578)

[1.2.12. ShaderUtil 79](#_Toc6871579)

[1.2.13. GridAxisShader 81](#_Toc6871580)

[1.2.14. Primitive2DShader 81](#_Toc6871581)

[1.2.15. Transform 82](#_Toc6871582)

[1.3. Вспомогательный модуль 83](#_Toc6871583)

[1.3.1. Colour4f 83](#_Toc6871584)

[1.3.2. Mat3f 86](#_Toc6871585)

[1.3.3. Mat4f 91](#_Toc6871586)

[1.3.4. Utils 95](#_Toc6871587)

[1.3.5. Vec2f 97](#_Toc6871588)

[1.3.6. Vec3f 103](#_Toc6871589)

[ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ 115](#_Toc6871590)

[ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ 116](#_Toc6871591)

# АННОТАЦИЯ

В данном документе приведет текст курсовой работы «Библиотека инверсной кинематики в web-приложениях», результатом которой является библиотека, написанная на языке программирования TypeScript.

Библиотека предназначена для предоставления возможности работы со скелетной анимацией, без необходимости реализовывать алгоритм инверсной кинематики самостоятельно и устанавливать какие-либо специальные программные средства.

Настоящий документ разработан в соответствии с требованиями:

1) ГОСТ 19.103-77 Обозначения программ и программных документов [1];

2) ГОСТ 19.104-78 Основные надписи [2];

3) ГОСТ 19.105-78 Общие требования к программным документам [3];

4) ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом [4];

5) ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению [5].

Изменения к данному документу оформляются согласно ГОСТ 19.603-78 [6], ГОСТ 19.604-78 [7].

# ТЕКСТ ПРОГРАММЫ

Библиотека состоит из трех модулей:

* аналитический модуль;
* модуль визуализации;
* вспомогательный модуль.

Каждый мз модулей содержит 13, 16 и 6 классов соответственно.

## Аналитический модуль

### BaseboneConstraintType2D

*/\*\*  
 \* Типы ограничений первой кости в цепи.  
 \*  
 \* Имеет следующие значения:  
 \* <ul>  
 \* <li>NONE - Без ограничений.</li>  
 \* <li>GLOBAL\_ABSOLUTE - Ограничена относительно мировых координат.</li>  
 \* <li>LOCAL\_RELATIVE - Ограничена относительно прикреплённой к ней кости.</li>  
 \* <li>LOCAL\_ABSOLUTE - Ограничена относительно прикреплённой к ней кости.</li>  
 \* </ul>  
 \*/***export enum** BaseboneConstraintType2D { *NONE*, *GLOBAL\_ABSOLUTE*, *LOCAL\_RELATIVE*, *LOCAL\_ABSOLUTE* }

### BaseboneConstraintType3D

*/\*\*  
 \* Типы ограничений первой кости в цепи.  
 \*  
 \* Имеет следующие значения:  
 \* <ul>  
 \* <li>NONE - Без ограничений.</li>  
 \* <li>GLOBAL\_ROTOR - Ограничена относительно мировых координат. Имеет шарообразные возможности вращения.</li>  
 \* <li>LOCAL\_ROTOR - Ограничена относительно прикреплённой к ней кости. Имеет шарообразные возможности вращения.</li>  
 \* <li>GLOBAL\_HINGE - Ограничена относительно мировых координат. Имеет шарнирные возможности вращения. </li>  
 \* <li>LOCAL\_HINGE - Ограничена относительно прикреплённой к ней кости. Имеет шарнирные возможности вращения. </li>  
 \* </ul>  
 \*/***export enum** BaseboneConstraintType3D {  
 *NONE*,  
 *GLOBAL\_ROTOR*,  
 *LOCAL\_ROTOR*,  
 *GLOBAL\_HINGE*,  
 *LOCAL\_HINGE*}

### BoneConnectionPoint

*/\*\*  
 \* Типы связи между FabrikBone.  
 \* В начале или конце кости хозяина располагается данная кость.  
 \*/***export enum** BoneConnectionPoint { *START*, *END* }

### ConstraintCoordinateSystem

*/\*\*  
 \* ConstraintCoordinateSystem Система координат, относительно которой рассматривается ограничения.  
 \* <p>  
 \* LOCAL (default) - относительно предыдущей кости в цепи.  
 \* GLOBAL - относитьльно мировой системы координат.  
 \*/***export enum** ConstraintCoordinateSystem { *LOCAL*, *GLOBAL* }

### FabrikBone2D

**import** {Vec2f} **from '../utils/Vec2f'**;  
**import** {FabrikJoint2D} **from './FabrikJoint2D'**;  
**import** {Colour4f} **from '../utils/Colour4f'**;  
**import** {Utils} **from '../utils/Utils'**;  
**import** {ConstraintCoordinateSystem} **from './ConstraintCoordinateSystem'**;  
  
*/\*\*  
 \* Кость.  
 \* <p>  
 \* FabrikBone2D состоит из начальной и конечной позиции и FabrikJoint2D, который ограничивает возможность поворота  
 \* относительно предыдущей кости в цепи или мировой системы координат.  
 \*/***export class** FabrikBone2D {  
 */\*\*  
 \* \_joint Сустав прикреплённый к FabrikBone2D.  
 \* <p>  
 \* Каждая кость содержит один сустав, который крепит данную кость к предыдущей.  
 \* <p>  
 \* По умолчанию сустав ничем не ограничен. Он может вращаться во всех возможных направлениях.  
 \* Изменить параметры сустава можно через конструктор кости или с помощью методов  
 \* {****@link*** *#setClockwiseConstraintDegs(float)} и {****@link*** *#setAnticlockwiseConstraintDegs(float)}.  
 \*/* **private \_joint** = **new** FabrikJoint2D();  
  
 */\*\*  
 \* \_startLocation Начальная позиция кости.  
 \* <p>  
 \* Начальная позиция может быть задана через конструктоp или с помощью методов 'addBone'  
 \* или 'addConsecutiveBone' из класса {****@link*** *FabrikChain2D}.  
 \*/* **private \_startLocation** = **new** Vec2f();  
  
 */\*\*  
 \* \_endLocation Конечная позиция кости.  
 \* <p>  
 \* Конечная позиция может быть задана через конструктоp или с помощью метода 'addBone'  
 \* из класса {****@link*** *FabrikChain2D}.  
 \*/* **private \_endLocation** = **new** Vec2f();  
  
 */\*\*  
 \* \_name Название кости.  
 \* <p>  
 \* Необязательное поле,облегчающее идентификацию кости в цепи.  
 \*/* **private \_name**: **string** = **''**;  
  
 */\*\*  
 \* \_length Длина кости вычисленная из её начальной и конечной позиции.  
 \* <p>  
 \* В процессе использования кости её длина не меняется.  
 \*/* **private \_length**: **number**;  
  
 **private \_globalConstraintUV** = **new** Vec2f(1.0, 0.0);  
  
 */\*\*  
 \* Цвет кости.  
 \* <p>  
 \* По умолчанию кость белая непрозрачная.  
 \*/* **private \_colour** = **new** Colour4f();  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает несколько возможных комбинаций параметров:  
 \* <p>  
 \* Конструктор без параметров. Все поля класса принимают значения по умолчанию.  
 \* <p>  
 \* Если третий параметр конструктора задан как null или undefiened, но при этом первые 2 параметра присутствуют,  
 \* то конструктор будет воспринимать первые 2 параметра как начальную и конечную позиции данной кости.  
 \* Параметры с четвертого по седьмой опциональны.  
 \*  
 \** ***@param*** *startLocation Начальная позиция кости в мировой системе координат.  
 \** ***@param*** *endLocationOrDirection Конечная позиция кости в мировой системе координат.  
 \** ***@param*** *length null или undefined.  
 \** ***@param*** *cwConstraintDegs Угол-ограничение по часовой стрелке.  
 \** ***@param*** *acwConstraintDegs Угол-ограничение против часовой стрелки.  
 \** ***@param*** *colour Цвет кости.  
 \** ***@param*** *name Название кости.  
 \* <p>  
 \* Если третий параметр конструктора - это число, но при этом первые 2 параметра присутствуют,  
 \* то конструктор будет воспринимать первые 2 параметра как начальную позицию данной кости и её направление.  
 \* Параметры с четвертого по седьмой опциональны.  
 \*  
 \** ***@param*** *startLocation Начальная позиция кости в мировой системе координат.  
 \** ***@param*** *endLocationOrDirection Конечная позиция кости в мировой системе координат.  
 \** ***@param*** *length null или undefined.  
 \** ***@param*** *cwConstraintDegs Угол-ограничение по часовой стрелке.  
 \** ***@param*** *acwConstraintDegs Угол-ограничение против часовой стрелки.  
 \** ***@param*** *colour Цвет кости.  
 \** ***@param*** *name Название кости.  
 \*/* **constructor**(  
 startLocation?: Vec2f,  
 endLocationOrDirection?: Vec2f,  
 length?: **number** | **null**,  
 cwConstraintDegs?: **number**,  
 acwConstraintDegs?: **number**,  
 colour?: Colour4f,  
 name?: **string**,  
 ) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **break**;  
 **case** 1:  
 **throw new** Error(**'FabrikBone2D constructor can not have 1 param.'**);  
 **default**:  
 **if** (!startLocation || !endLocationOrDirection) {  
 **throw new** Error(**'Invalid FabrikBone2D constructor params.'**);  
 }  
 **if** (length === **null** || length === **undefined**) {  
 *// Представляем первые 2 вектора как начальную и конечную позиции кости.* **this**.**\_startLocation**.set(startLocation);  
 **this**.**\_endLocation**.set(endLocationOrDirection);  
 **this**.\_updateLength();  
 } **else** {  
 *// Представляем первые 2 вектора как начальную позицию и направление кости.* Utils.*validateDirectionUV*(endLocationOrDirection);  
  
 **this**.**\_startLocation**.set(startLocation);  
 **this**.**\_endLocation**.set( **this**.**\_startLocation**.*plus*( Vec2f.*normalised*(endLocationOrDirection).*times*(length) ) );  
  
 **this**.\_updateLength();  
 }  
  
 **if** (**typeof** cwConstraintDegs === **'number'**) **this**.setClockwiseConstraintDegs(cwConstraintDegs);  
 **if** (**typeof** acwConstraintDegs === **'number'**) **this**.setAnticlockwiseConstraintDegs(acwConstraintDegs);  
 **if** (colour **instanceof** Colour4f) **this**.**\_colour**.set(colour.*toArray*());  
 **if** (**typeof** name === **'string'**) **this**.**\_name** = name;  
 }  
 }  
  
 *// ---------- Methods ----------  
  
 /\*\*  
 \* Задаёт всем полям данной кости такие жа значения как у кости источника.  
 \* <p>  
 \* Копирование свойств неглубокое.  
 \* После применнеия метода данная кость и кость источник будут содержать общие элементы.  
 \*  
 \** ***@param*** *source Кость источник.  
 \*/* set(source: FabrikBone2D) {  
 **this**.**\_startLocation**.set(source.**\_startLocation**);  
 **this**.**\_endLocation**.set(source.**\_endLocation**);  
 **this**.**\_joint**.set(source.**\_joint**);  
 **this**.**\_colour**.set(source.**\_colour**.*toArray*());  
  
 **this**.**\_name** = source.**\_name**;  
 **this**.**\_length** = source.**\_length**;  
 **this**.**\_globalConstraintUV** = source.**\_globalConstraintUV**;  
 }  
  
 *getGlobalConstraintUV* = () => **this**.**\_globalConstraintUV**;  
  
 */\*\*  
 \* Возвращает длину кости.  
 \*  
 \** ***@return*** *\*/  
 getLength* = (): **number** => **this**.**\_length**;  
  
 */\*\*  
 \* Задаёт цвет кости.  
 \*  
 \** ***@param*** *colour Цвет кости.  
 \*/* setColour(colour: Colour4f): **void** {**this**.**\_colour**.set(colour.*toArray*()); }  
  
 */\*\*  
 \* Возвращает цвет кости.  
 \*  
 \** ***@return*** *Цвет кости.  
 \*/  
 getColour* = (): Colour4f => **this**.**\_colour**;  
  
 */\*\*  
 \* Задаёт начальную позицию кости.  
 \*  
 \** ***@param*** *location Начальная позиция кости.  
 \*/* setStartLocation(location: Vec2f): **void** {  
 **this**.**\_startLocation**.set(location);  
 }  
  
 */\*\*  
 \* Возвращает начальную позицию кости.  
 \*  
 \** ***@return*** *Начальная позиция кости.  
 \*/  
 getStartLocation* = (): Vec2f => **this**.**\_startLocation**;  
  
 */\*\*  
 \* Возвращает начальную позицию кости в виде числового массива.  
 \*  
 \** ***@return*** *Начальная позиция кости.  
 \*/  
 getStartLocationAsArray* = (): **number**[] => [**this**.**\_startLocation**.**x**, **this**.**\_startLocation**.**y**];  
  
 */\*\*  
 \* Задаёт конечную позицию кости.  
 \*  
 \** ***@param*** *location Конечная позиция кости.  
 \*/* setEndLocation(location: Vec2f): **void** {  
 **this**.**\_endLocation**.set(location);  
 }  
  
 */\*\*  
 \* Возвращает конечную позицию кости.  
 \*  
 \** ***@return*** *Конечная позиция кости.  
 \*/  
 getEndLocation* = (): Vec2f => **this**.**\_endLocation**;  
  
 */\*\*  
 \* Возвращает конечную позицию кости в виде числового массива.  
 \*  
 \** ***@return*** *Конечная позиция кости.  
 \*/  
 getEndLocationAsArray* = (): **number**[] => [**this**.**\_endLocation**.**x**, **this**.**\_endLocation**.**y**];  
  
 */\*\*  
 \* Задаёт сустав для кости.  
 \*  
 \** ***@param*** *joint Сустав.  
 \*/* setJoint(joint: FabrikJoint2D): **void** { **this**.**\_joint**.set(joint); }  
  
 */\*\*  
 \* Возвращает сустав для кости.  
 \*  
 \** ***@return*** *Сустав.  
 \*/  
 getJoint* = (): FabrikJoint2D => **this**.**\_joint**;  
  
 */\*\*  
 \* Задаёт угол-ограничение по часовой стрелке для сустава кости.  
 \* <p>  
 \* При передаче угла вне допустимого диапазона, угол приравневается к одному из граничных значений.  
 \*  
 \** ***@param*** *angleDegs Угол-ограничение в градусах.  
 \*/* setClockwiseConstraintDegs(angleDegs: **number**): **void** { **this**.**\_joint**.setClockwiseConstraintDegs(angleDegs); }  
  
 */\*\*  
 \* Возвращает угол-ограничение по часовой стрелке для сустава кости.  
 \*  
 \** ***@return*** *Угол-ограничение по часовой стрелке в градусах.  
 \*/  
 getClockwiseConstraintDegs* = (): **number** => **this**.**\_joint**.*getClockwiseConstraintDegs*();  
  
 */\*\*  
 \* Задаёт угол-ограничение против часовой стрелки для сустава кости.  
 \* <p>  
 \* При передаче угла вне допустимого диапазона, угол приравневается к одному из граничных значений.  
 \*  
 \** ***@param*** *angleDegs Угол-ограничение в градусах.  
 \*/* setAnticlockwiseConstraintDegs(angleDegs: **number**): **void** { **this**.**\_joint**.setAnticlockwiseConstraintDegs(angleDegs); }  
  
 */\*\*  
 \* Возвращает угол-ограничение против часовой стрелки для сустава кости.  
 \*  
 \** ***@return*** *Угол-ограничение против часовой стрелки в градусах.  
 \*/* getAnticlockwiseConstraintDegs(): **number** { **return this**.**\_joint**.*getAnticlockwiseConstraintDegs*(); }  
  
 */\*\*  
 \* Возвращает направление кости.  
 \* <p>  
 \* Если необходимо получить направление обратное направлению кости - воспользуйтесь методом negated вектора.  
 \*  
 \** ***@return*** *Единичный вектор направления данной кости.  
 \** ***@see*** *Vec2f#negated()  
 \*/  
 getDirectionUV* = (): Vec2f => Vec2f.*getDirectionUV*(**this**.**\_startLocation**, **this**.**\_endLocation**);  
  
 */\*\*  
 \* Задаёт значение имени кости.  
 \*  
 \** ***@param*** *name Имя.  
 \*/* setName(name: **string**): **void** { **this**.**\_name** = name; }  
  
 */\*\*  
 \* Возвращает имя кости.  
 \*  
 \** ***@return*** *Имя.  
 \*/  
 getName* = (): **string** => **this**.**\_name**;  
  
 */\*\*  
 \* Задаёт тип сустава.  
 \*  
 \** ***@param*** *coordSystem Тип сустава.  
 \*/* setJointConstraintCoordinateSystem(coordSystem: ConstraintCoordinateSystem): **void** {  
 **this**.**\_joint**.*setConstraintCoordinateSystem*(coordSystem);  
 }  
  
 */\*\*  
 \* Возвращает тип сустава.  
 \*  
 \** ***@return*** *Тип сустава.  
 \*/  
 getJointConstraintCoordinateSystem* = (): ConstraintCoordinateSystem => **this**.**\_joint**.*getConstraintCoordinateSystem*();  
  
 */\*\*  
 \* Пересчитывает длину кости, опираясь на внутренние поля начальной и конечной позиции.  
 \*/* **private** \_updateLength(): **void** {  
 **const** length = Vec2f.*distanceBetween*(**this**.**\_startLocation**, **this**.**\_endLocation**);  
  
 **if** (length >= 0.0) {  
 **this**.**\_length** = length;  
 }  
 **else** {  
 **throw new** Error(**'Bone getLength must be a positive value.'**);  
 }  
 }  
}

### FabrikBone3D

**import** {BoneConnectionPoint} **from './BoneConnectionPoint'**;  
**import** {FabrikJoint3D} **from './FabrikJoint3D'**;  
**import** {JointType} **from './JointType'**;  
**import** {Vec3f} **from '../utils/Vec3f'**;  
**import** {Colour4f} **from '../utils/Colour4f'**;  
**import** {Utils} **from '../utils/Utils'**;  
  
*/\*\*  
 \* Кость.  
 \* <p>  
 \* FabrikBone3D состоит из начальной и конечной позиции и FabrikJoint3D, который ограничивает возможность поворота.  
 \*/***export class** FabrikBone3D {  
 */\*\*  
 \* Если данная кость присоединена к кости из другой цепи, к какому концу другой цепи она должна присоединяться?  
 \* <p>  
 \* По умолчанию присоединяется к концу данной кости.  
 \* <p>  
 \* Это поле может быть изменено с помощью метода {#link #setBoneConnectionPoint(BoneConnectionPoint)}, или при соединении кости  
 \* с другой цепью с помощью метода {****@link*** *FabrikStructure3D#connectChain(FabrikChain3D, number, number, BoneConnectionPoint)}.  
 \*/* **private \_boneConnectionPoint**: BoneConnectionPoint = BoneConnectionPoint.*END*;  
  
 */\*\*  
 \* \_joint Сустав для данной кости.  
 \* <p>  
 \* У каждой кости есть один сустав, располагающийся в начальной позиции кости.  
 \* Он определяет как кость может быть повернута относительно предыдущей кости в цепи.  
 \* <p>  
 \* По умолчанию сустав не имеет ограничений, но они могут быть заданы с помощью методов  
 \* {****@link*** *#setClockwiseConstraintDegs(number)} и {****@link*** *#setAnticlockwiseConstraintDegs(number)}.  
 \*/* **private \_joint**: FabrikJoint3D = **new** FabrikJoint3D();  
  
 */\*\*  
 \* \_startLocation Начальная позиция кости.  
 \* <p>  
 \* Начальная позиция должна задаваться только через конструктор или  
 \* с помощью метода 'addBone' класса {****@link*** *FabrikChain3D}.  
 \*/* **private \_startLocation**: Vec3f = **new** Vec3f();  
  
 */\*\*  
 \* \_endLocation Конечная позиция кости.  
 \* <p>  
 \* Конечная позиция должна задаваться только через конструктор  
 \* или через метод 'addBone' класса {****@link*** *FabrikChain3D}.  
 \*/* **private \_endLocation**: Vec3f = **new** Vec3f();  
  
 */\*\*  
 \* \_name Имя кости.  
 \* <p>  
 \* Не обязательно использовать это свойство.  
 \* Иногда его использование облегчает идентификацию кости в цепи.  
 \*  
 \** ***@see*** *#setName(string)  
 \** ***@see*** *#FabrikBone3D(Vec3f, Vec3f, string)  
 \** ***@see*** *#FabrikBone3D(Vec3f, Vec3f, number, string)  
 \*/* **private \_name**: **string** = **''**;  
  
 */\*\*  
 \* Длина кости.  
 \*/* **private \_length**: **number** = 0;  
  
 */\*\*  
 \* Цвет для визуализации кости.  
 \*  
 \** ***@default*** *Белый непрозрачный - [1, 1, 1, 1].  
 \*/* **private \_colour**: Colour4f = **new** Colour4f();  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает несколько возможных комбинаций параметров:  
 \* <p>  
 \* Конструктор без параметров. Все поля класса принимают значения по умолчанию.  
 \* <p>  
 \* Если третий параметр конструктора задан как null или undefiened, но при этом первые 2 параметра присутствуют,  
 \* то конструктор будет воспринимать первые 2 параметра как начальную и конечную позиции данной кости.  
 \* Параметры с четвертого по седьмой опциональны.  
 \*  
 \** ***@param*** *startLocation Начальная позиция кости в мировой системе координат.  
 \** ***@param*** *endLocationOrDirection Конечная позиция кости в мировой системе координат.  
 \** ***@param*** *length null или undefined.  
 \** ***@param*** *colour Цвет кости.  
 \** ***@param*** *name Название кости.  
 \* <p>  
 \* Если третий параметр конструктора - это число, но при этом первые 2 параметра присутствуют,  
 \* то конструктор будет воспринимать первые 2 параметра как начальную позицию данной кости и её направление.  
 \* Параметры с четвертого по седьмой опциональны.  
 \*  
 \** ***@param*** *startLocation Начальная позиция кости в мировой системе координат.  
 \** ***@param*** *endLocationOrDirection Конечная позиция кости в мировой системе координат.  
 \** ***@param*** *length null или undefined.  
 \** ***@param*** *colour Цвет кости.  
 \** ***@param*** *name Название кости.  
 \*/* **constructor**(  
 startLocation?: Vec3f,  
 endLocationOrDirection?: Vec3f,  
 length?: **number** | **null**,  
 colour?: Colour4f,  
 name?: **string**,  
 ) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **break**;  
 **case** 1:  
 **throw new** Error(**'FabrikBone2D constructor can not have 1 param.'**);  
 **default**:  
 **if** (!startLocation || !endLocationOrDirection) {  
 **throw new** Error(**'Invalid FabrikBone2D constructor params.'**);  
 }  
 **if** (length === **null** || length === **undefined**) {  
 *// Представляем первые 2 вектора как начальную и конечную позиции кости.* **this**.**\_startLocation**.set(startLocation);  
 **this**.**\_endLocation**.set(endLocationOrDirection);  
 **this**.\_updateLength();  
 } **else** {  
 *// Представляем первые 2 вектора как начальную позицию и направление кости.* Utils.*validateDirectionUV*(endLocationOrDirection);  
  
 **this**.**\_startLocation**.set(startLocation);  
 **this**.**\_endLocation**.set( **this**.**\_startLocation**.*plus*( endLocationOrDirection.normalised().times(length) ) );  
  
 **this**.\_updateLength();  
 }  
  
 **if** (colour **instanceof** Colour4f) **this**.**\_colour**.set(colour.*toArray*());  
 **if** (**typeof** name === **'string'**) **this**.**\_name** = name;  
 }  
 }  
  
 *// ---------- Methods ----------  
 /\*\*  
 \* Задаёт всем полям данной кости такие жа значения как у кости источника.  
 \* <p>  
 \* Копирование свойств неглубокое.  
 \* После применнеия метода данная кость и кость источник будут содержать общие элементы.  
 \*  
 \** ***@param*** *source Кость источник.  
 \*/* set(source: FabrikBone3D) {  
 **this**.**\_startLocation**.set(source.**\_startLocation**);  
 **this**.**\_endLocation**.set(source.**\_endLocation**);  
 **this**.**\_joint**.set(source.**\_joint**);  
 **this**.**\_colour**.set(source.**\_colour**.*toArray*());  
  
 **this**.**\_name** = source.**\_name**;  
 **this**.**\_length** = source.**\_length**;  
 **this**.**\_boneConnectionPoint** = source.**\_boneConnectionPoint**;  
 }  
  
 */\*\*  
 \* Возвращает длину кости.  
 \*  
 \** ***@return*** *\*/  
 getLength* = (): **number** => **this**.**\_length**;  
  
 */\*\*  
 \* Возвращает длину кости.  
 \*  
 \** ***@return*** *\*/  
 getLifeLength* = (): **number** => Vec3f.*distanceBetween*(**this**.**\_startLocation**, **this**.**\_endLocation**);  
  
 */\*\*  
 \* Задаёт точку соединения.  
 \* <p>  
 \* Эта точка соединения контролирует куда присоединяется кость из другой цепи при соединении с нашей костью.  
 \* <p>  
 \* Значение по умолчанию BoneConnectionPoint3D.END.  
 \*  
 \** ***@param*** *bcp Точка соединения (BoneConnectionPoint3.START или BoneConnectionPoint.END).  
 \*  
 \*/* setBoneConnectionPoint(bcp: BoneConnectionPoint): **void** { **this**.**\_boneConnectionPoint** = bcp; }  
  
 */\*\*  
 \* Возвращает точку соединения с данной костью.  
 \*  
 \** ***@return*** *\*/  
 getBoneConnectionPoint* = (): BoneConnectionPoint => **this**.**\_boneConnectionPoint**;  
  
 */\*\*  
 \* Возвращает цвет кости  
 \*  
 \** ***@return*** *\*/  
 getColour* = (): Colour4f => **this**.**\_colour**;  
  
 */\*\*  
 \* Задаёт цвет кости.  
 \*  
 \** ***@param*** *colour Будущий цвет кости для отрисовки.  
 \*/* setColour(colour: Colour4f): **void** { **this**.**\_colour**.set(colour.*toArray*()); }  
  
 */\*\*  
 \* Возвращает начальную позицию кости.  
 \*  
 \** ***@return*** *Начальная позиция кости.  
 \*/  
 getStartLocation* = (): Vec3f => **this**.**\_startLocation**;  
  
 */\*\*  
 \* Возвращает начальную позицию кости в виде числового массива.  
 \*  
 \** ***@return*** *Начальная позиция кости.  
 \*/  
 getStartLocationAsArray* = (): **number**[] => [**this**.**\_startLocation**.**x**, **this**.**\_startLocation**.**y**, **this**.**\_startLocation**.**z**];  
  
 */\*\*  
 \* Возвращает конечную позицию кости.  
 \*  
 \** ***@return*** *Конечная позиция кости.  
 \*/  
 getEndLocation* = (): Vec3f => **this**.**\_endLocation**;  
  
 */\*\*  
 \* Возвращает конечную позицию кости в виде числового массива.  
 \*  
 \** ***@return*** *Конечная позиция кости.  
 \*/  
 getEndLocationAsArray* = (): **number**[] => [**this**.**\_endLocation**.**x**, **this**.**\_endLocation**.**y**, **this**.**\_endLocation**.**z**];  
  
 */\*\*  
 \* Задаёт сустав для кости.  
 \*  
 \** ***@param*** *joint Сустав.  
 \*/* setJoint(joint: FabrikJoint3D): **void** { **this**.**\_joint**.set(joint); }  
  
 */\*\*  
 \* Возвращает сустав для кости.  
 \*  
 \** ***@return*** *Сустав.  
 \*/  
 getJoint* = (): FabrikJoint3D => **this**.**\_joint**;  
  
 */\*\*  
 \* Возвращает тип сустава.  
 \*  
 \** ***@return*** *\*/  
 getJointType* = (): JointType => **this**.**\_joint**.getJointType();  
  
 */\*\*  
 \* Задаёт угол ограничение по часовой стрелке для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибки, если угол выходит за границы допустимых значений и  
 \* если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *angleDegs Угол ограничение по часовой стрелке в градусах.  
 \*/* setHingeJointClockwiseConstraintDegs(angleDegs): **void** { **this**.**\_joint**.setHingeJointClockwiseConstraintDegs(angleDegs); }  
  
 */\*\*  
 \* Возвращает угол-ограничение по часовой стрелке для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шарнирный.  
 \*  
 \** ***@return*** *Угол-ограничение по часовой стрелке в градусах.  
 \*/* getHingeJointClockwiseConstraintDegs(): **number** { **return this**.**\_joint**.getHingeClockwiseConstraintDegs(); }  
  
 */\*\*  
 \* Задаёт угол ограничение против часовой стрелки для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибки, если угол выходит за границы допустимых значений и  
 \* если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *angleDegs Угол ограничение против часовой стрелки в градусах.  
 \*/* setHingeJointAnticlockwiseConstraintDegs(angleDegs: **number**): **void** { **this**.**\_joint**.setHingeJointAnticlockwiseConstraintDegs(angleDegs); }  
  
 */\*\*  
 \* Возвращает угол-ограничение против часовой стрелки для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шарнирный.  
 \*  
 \** ***@return*** *Угол-ограничение против часовой стрелки в градусах.  
 \*/* getHingeJointAnticlockwiseConstraintDegs(): **number** { **return this**.**\_joint**.getHingeAnticlockwiseConstraintDegs(); }  
  
 */\*\*  
 \* Задаёт угол-ограничение для шарового сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шаровой или  
 \* угол выходит за границы допустимых значений.  
 \*  
 \** ***@param*** *angleDegs Угол-ограничение в градусах.  
 \*/* setBallJointConstraintDegs(angleDegs: **number**): **void** {  
 **this**.**\_joint**.setBallJointConstraintDegs(angleDegs);  
 }  
  
 */\*\*  
 \* Возвращает угол ограничение шарового сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шаровой  
 \*  
 \** ***@return*** *Угол-ограничение в градусах.  
 \*/* getBallJointConstraintDegs(): **number** { **return this**.**\_joint**.getBallJointConstraintDegs(); }  
  
 */\*\*  
 \* Возвращает направление кости.  
 \* <p>  
 \* Если необходимо получить направление обратное направлению кости - воспользуйтесь методом negated вектора.  
 \*  
 \** ***@return*** *Единичный вектор направления данной кости.  
 \** ***@see*** *Vec3f#negated()  
 \*/* getDirectionUV(): Vec3f {  
 **return** Vec3f.*getDirectionUV*(**this**.**\_startLocation**, **this**.**\_endLocation**);  
 }  
  
 */\*\*  
 \* Возвращает глобальный поворот кости относительно глобальной оси X.  
 \* Возвращаемое значение находится в диапазоне -179.9..180.0.  
 \*  
 \** ***@return*** *Отклонение вектора в градусах.  
 \*\*/* getGlobalPitchDegs(): **number** {  
 **return** Vec3f.*getDirectionUV*(**this**.**\_startLocation**, **this**.**\_endLocation**).getGlobalPitchDegs();  
 }  
  
 */\*\*  
 \* Возвращает глобальный поворот кости относительно глобальной оси Y.  
 \* Возвращаемое значение находится в диапазоне -179.9..180.0.  
 \*  
 \** ***@return*** *Отклонение вектора в градусах.  
 \*\*/* getGlobalYawDegs(): **number** {  
 **return** Vec3f.*getDirectionUV*(**this**.**\_startLocation**, **this**.**\_endLocation**).getGlobalYawDegs();  
 }  
  
 */\*\*  
 \* Задаёт имя кости.  
 \*  
 \** ***@param*** *name The name to set.  
 \*/* setName(name: **string**): **void** { **this**.**\_name** = name; }  
  
 */\*\*  
 \* Возвращает имя кости.  
 \*  
 \** ***@return*** *String  
 \*/  
 getName* = (): **string** => **this**.**\_name**;  
  
 */\*\*  
 \* Задаёт начальную позицию кости.  
 \*  
 \** ***@param*** *location Начальная позиция кости.  
 \*/* setStartLocation(location: Vec3f): **void** {  
 **this**.**\_startLocation**.set(location);  
 }  
  
 */\*\*  
 \* Задаёт конечную позицию кости.  
 \*  
 \** ***@param*** *location Конечная позиция кости.  
 \*/* setEndLocation(location: Vec3f): **void** {  
 **this**.**\_endLocation**.set(location);  
 }  
  
 */\*\*  
 \* Пересчитывает длину кости, опираясь на внутренние поля начальной и конечной позиции.  
 \*/* \_updateLength(): **void** {  
 **const** length = Vec3f.*distanceBetween*(**this**.**\_startLocation**, **this**.**\_endLocation**);  
  
 **if** (length >= 0.0) {  
 **this**.**\_length** = length;  
 } **else** {  
 **throw new** Error(**'Bone getLength must be a positive value.'**);  
 }  
 }  
}

### FabrikChain2D

**import** {FabrikBone2D} **from './FabrikBone2D'**;  
**import** {Vec2f} **from '../utils/Vec2f'**;  
**import** {BoneConnectionPoint} **from './BoneConnectionPoint'**;  
**import** {Colour4f} **from '../utils/Colour4f'**;  
**import** {Utils} **from '../utils/Utils'**;  
**import** {ConstraintCoordinateSystem} **from './ConstraintCoordinateSystem'**;  
**import** {BaseboneConstraintType2D} **from './BaseboneConstraintType2D'**;  
  
*/\*\*  
 \* 2D цепь костей, решающая задачу инверсной кинематики с помощью алгоритма FABRIK.  
 \*/***export class** FabrikChain2D {  
  
 *// ---------- Private Properties ----------  
  
 /\*\*  
 \* Массив костей.  
 \*/* **private \_chain**: FabrikBone2D[] = [];  
  
 */\*\*  
 \* Название цепи.  
 \* <p>  
 \* Необязательное к использованию поле, помогающее идентифицировать цепь.  
 \*/* **private \_name**: **string** = **''**;  
   
 */\*\*  
 \* Порог, достигнув которого мы считаем задачу инверсной кинематики решенной.  
 \* <p>  
 \* При решении задачи ИК, когда расстояние между текущим положением и желаемым положением становится меньше или равно  
 \* значению порога, мы прерываем дальнейшие итерации и считаем задачу решенной.  
 \* <p>  
 \* Значение по умолчанию 1.  
 \* <p>  
 \* Минимально допустимое значение - 0, но стоит помнить о погрешности вычисления и не задавать порог равным нулю, так  
 \* как это может привести к бесконечным бесполезным попыткам.  
 \* <p>  
 \* Хотя данное поле и самый важный критерий останова для ИК-алгоритма, оно работает в комбинации с  
 \* {****@link*** *#\_maxIterationAttempts} и {****@link*** *\_minIterationChange}. Эти вспомогательные свойства помогают пережить ситуации,  
 \* когда длины цепи не хватает для достижения цели или когда суставы цепи сильно ограничены.  
 \*/* **private \_solveDistanceThreshold**: **number** = 1.0;  
  
  
 */\*\*  
 \* Максимальное количество итераций алгоритма, применяемых при решении задачи ИК.  
 \*  
 \** ***@default*** *20  
 \*/* **private \_maxIterationAttempts**: **number** = 15;  
  
 */\*\*  
 \* Минимальное значения для дельты рассотояний между итерациями алгоритма.  
 \*  
 \** ***@default*** *0.01  
 \*/* **private \_minIterationChange**: **number** = 0.01;  
  
 */\*\*  
 \* Длина цепи.  
 \*/* **private \_chainLength**: **number** = 0;  
  
 */\*\*  
 \* Начальная позиция первой кости в цепи.  
 \*  
 \** ***@default:*** *Vec3f(0, 0)  
 \*/* **private \_baseLocation**: Vec2f = **new** Vec2f();  
  
 */\*\*  
 \* Зафиксирована ли начальная позиция цепи.  
 \* Если нет - то она может перемещаться.  
 \*  
 \** ***@default*** *true  
 \*/* **private \_fixedBaseMode**: **boolean** = **true**;  
  
 */\*\*  
 \* Тип сустава первой кости в цепи.  
 \*/* **private \_baseboneConstraintType**: BaseboneConstraintType2D = BaseboneConstraintType2D.*NONE*;  
  
 **private \_boneConnectionPoint**: BoneConnectionPoint = BoneConnectionPoint.*END*;  
  
 */\*\*  
 \* Направление относительно которого мы ограничиваем первую кость цепи.  
 \*/* **private \_baseboneConstraintUV**: Vec2f = **new** Vec2f();  
  
 **private \_baseboneRelativeConstraintUV**: Vec2f = **new** Vec2f();  
  
 */\*\*  
 \* Последняя целевая позиция, относительно которой решалась задача ИК.  
 \*  
 \** ***@default*** *Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE)  
 \*/* **private \_lastTargetLocation**: Vec2f = **new** Vec2f(Number.MAX\_VALUE, Number.MAX\_VALUE);  
  
 */\*\*  
 \* Предыдущая начальная позиция базовой (первой в цепи) кости.  
 \* <p>  
 \* Данное значение хранится для того, чтобы сравнивать изменилось ли  
 \* начальное положение базовой кости в процессе решения.  
 \*  
 \** ***@default*** *Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE)  
 \*/* **private \_lastBaseLocation**:Vec2f = **new** Vec2f(Number.MAX\_VALUE, Number.MAX\_VALUE);  
  
 */\*\*  
 \* Встроенное целевое расположение,которое можно использовать для решения ИК.  
 \* <p>  
 \* Встроенные целевые местоположения позволяют решать ИК структуры для нескольких целей (по одной на цепочку в структуре)  
 \* вместо того, чтобы все цепочки решались для одной и той же цели. Чтобы использовать встроенные цели, флаг \_useEmbeddedTargets  
 \* должно быть true(что не является значением по умолчанию) - этот флаг можно установить с помощью вызова setEmbeddedTargetMode (true).  
 \*/* **private \_embeddedTarget**: Vec2f = **new** Vec2f();  
  
 */\*\*  
 \* \_useEmbeddedTarget Следует ли использовать местоположение \_embeddedTarget при решении этой цепочки.  
 \* <p>  
 \* Этот флаг можно переключить, вызвав setEmbeddedTargetMode (true) в цепочке.  
 \*  
 \** ***@default*** *false  
 \*/* **private \_useEmbeddedTarget**: **boolean** = **false**;  
  
 */\*\*  
 \* Текущее расстояние между эффектором(концом последней точки в цепи) и целевой позицией.  
 \*/* **private \_currentSolveDistance**: **number** = Number.MAX\_VALUE;  
  
 */\*\*  
 \* Номер цепи, к которой эта цепь подключена в FabrikStructure3D.  
 \* <p>  
 \* Если значение -1, значит данная цепь ни с чем не связана.  
 \*  
 \** ***@default*** *-1  
 \*/* **private \_connectedChainNumber**: **number** = -1;  
  
 */\*\*  
 \* Номер кости, с которой связана эта цепь, если она вообще связана с другой цепью.  
 \* <p>  
 \* Если значение -1, значит данная цепь ни с чем не связана.  
 \*  
 \** ***@default*** *-1  
 \*/* **private \_connectedBoneNumber**: **number** = -1;  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \*  
 \* Принимает три вариации параметров:  
 \*  
 \* Без параметров. Всем полям проставляются значения по умолчанию.  
 \*  
 \* Один параметр-строка.  
 \*  
 \** ***@param*** *source Имя данной цепи.  
 \*  
 \* Конструктор копирования.  
 \*  
 \** ***@param*** *source Цепь для копирования.  
 \*/* **constructor**(source?: **string** | FabrikChain2D) {  
 **if** (!source) {  
 **return**;  
 }  
 **if** (**typeof** source === **'string'**) {  
 **this**.**\_name** = source;  
 } **else** {  
 **this**.**\_chain** = source.cloneChainVector();  
 **this**.**\_baseLocation**.set(source.**\_baseLocation**);  
 **this**.**\_lastTargetLocation**.set(source.**\_lastTargetLocation**);  
 **this**.**\_lastBaseLocation**.set(source.**\_lastBaseLocation**);  
 **this**.**\_baseboneConstraintUV**.set(source.**\_baseboneConstraintUV**);  
 **this**.**\_baseboneRelativeConstraintUV**.set(source.**\_baseboneRelativeConstraintUV**);  
 **this**.**\_embeddedTarget**.set(source.**\_embeddedTarget**);  
  
 **this**.**\_chainLength** = source.**\_chainLength**;  
 **this**.**\_currentSolveDistance** = source.**\_currentSolveDistance**;  
 **this**.**\_connectedChainNumber** = source.**\_connectedChainNumber**;  
 **this**.**\_connectedBoneNumber** = source.**\_connectedBoneNumber**;  
 **this**.**\_baseboneConstraintType** = source.**\_baseboneConstraintType**;  
 **this**.**\_boneConnectionPoint** = source.**\_boneConnectionPoint**;  
 **this**.**\_name** = source.**\_name**;  
 **this**.**\_useEmbeddedTarget** = source.**\_useEmbeddedTarget**;  
 }  
 }  
  
*// ---------- Public Methods ------------  
  
 /\*\*  
 \* Добавляет кость в конец цепи.  
 \*  
 \** ***@param*** *bone Кость.  
 \*/* addBone(bone: FabrikBone2D): **void** {  
 **this**.**\_chain**.push(bone);  
  
 **if** (**this**.**\_chain**.**length** === 1) {  
 **this**.**\_baseLocation**.set( bone.*getStartLocation*() );  
  
 **this**.**\_baseboneConstraintUV** = bone.*getDirectionUV*();  
 }  
  
 **this**.updateChainLength();  
 }  
  
 addConsecutiveConstrainedBone(directionUV: Vec2f, length: **number**, clockwiseDegs: **number**, anticlockwiseDegs: **number**, colour: Colour4f = **new** Colour4f()): **void** {  
 Utils.*validateDirectionUV*(directionUV);  
  
 Utils.*validateLength*(length);  
  
 **if** (**this**.**\_chain**.**length** !== 0) {  
 **const** prevBoneEnd = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 **let** bone = **new** FabrikBone2D(prevBoneEnd, Vec2f.*normalised*(directionUV), length, clockwiseDegs, anticlockwiseDegs, colour);  
 **this**.addBone(bone);  
 } **else** {  
 **throw new** Error(**'You cannot add the base bone to a chain using this method as it does not provide a start location.'**);  
 }  
 }  
  
 addConsecutiveBone(directionUV: Vec2f, length: **number**): **void** {  
 **this**.addConsecutiveConstrainedBone( directionUV, length, 180.0, 180.0, **new** Colour4f() );  
 }  
  
 addConsecutiveCreatedBone(bone: FabrikBone2D): **void** {  
 **const** dir = bone.*getDirectionUV*();  
 Utils.*validateDirectionUV*(dir);  
  
 **const** len = bone.*getLength*();  
 Utils.*validateLength*(len);  
  
 **if** (**this**.**\_chain**.**length** !== 0) {  
 **const** prevBoneEnd = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 bone.setStartLocation(prevBoneEnd);  
 bone.setEndLocation( prevBoneEnd.*plus*(dir.*times*(len)) );  
  
 **this**.addBone(bone);  
 } **else** {  
 **throw new** Error(**'You cannot add the base bone to a chain using this method as it does not provide a start location.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает тип сустава базовой кости.  
 \*  
 \** ***@return*** *\*/* getBaseboneConstraintType(): BaseboneConstraintType2D { **return this**.**\_baseboneConstraintType**; }  
  
 getBaseboneConstraintUV(): Vec2f { **return this**.**\_baseboneConstraintUV**; }  
  
 */\*\*  
 \* Возвращает базовое расположение цепочки IK.  
 \* <p>  
 \* Независимо от того, сколько костей содержится в цепи, базовое местоположение всегда является начальным местоположением  
 \* первой кости в цепи.  
 \*  
 \** ***@return*** *\*/* getBaseLocation(): Vec2f {  
 **if** (**this**.**\_chain**.**length** !== 0) {  
 **return this**.**\_chain**[0].*getStartLocation*();  
 } **else** {  
 **throw new** Error(**'Cannot get base location as there are zero bones in the chain.'**);  
 }  
 }  
  
 getBone(boneNumber: **number**): FabrikBone2D {  
 **return this**.**\_chain**[boneNumber];  
 }  
  
 getBoneConnectionPoint(): BoneConnectionPoint { **return this**.**\_boneConnectionPoint**; }  
  
 getChain(): FabrikBone2D[] { **return this**.**\_chain**; }  
  
 getChainLength(): **number** { **return this**.**\_chainLength**; }  
  
 getConnectedBoneNumber(): **number** { **return this**.**\_connectedBoneNumber**; }  
  
 getConnectedChainNumber(): **number** { **return this**.**\_connectedChainNumber**; }  
  
 getEffectorLocation(): Vec2f {  
 **if** (**this**.**\_chain**.**length** !== 0) {  
 **return this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
 }  
 **else** {  
 **throw new** Error(**'Cannot get effector location as there are zero bones in the chain.'**);  
 }  
 }  
  
 getEmbeddedTargetMode(): **boolean** { **return this**.**\_useEmbeddedTarget**; }  
  
 getEmbeddedTarget(): Vec2f { **return this**.**\_embeddedTarget**; }  
  
 getLastTargetLocation(): Vec2f { **return this**.**\_lastTargetLocation**; }  
  
 getName(): **string** { **return this**.**\_name**; }  
  
 getNumBones(): **number** { **return this**.**\_chain**.**length**; }  
  
 */\*\*  
 \* Удаляет кость из этой цепи IK по ее индексу в цепи.  
 \* <p>  
 \* Если номер кости, которую необходимо удалить, не существует в цепочке, то возникает ошибка.  
 \*  
 \** ***@param*** *boneNumber Индекс кости в цепи.  
 \*/* removeBone(boneNumber: **number**): **void** {  
 **if** (boneNumber < **this**.**\_chain**.**length**) {  
 **this**.**\_chain**.splice(boneNumber, 1);  
 **this**.updateChainLength();  
 } **else** {  
 **throw new** Error(**'Bone '** + boneNumber + **' does not exist to be removed from the chain. Bones are zero indexed.'**);  
 }  
 }  
  
 setBaseboneConstraintType(type: BaseboneConstraintType2D): **void** { **this**.**\_baseboneConstraintType** = type; }  
  
 setBaseboneConstraintUV(constraintUV: Vec2f): **void** {  
 Utils.*validateDirectionUV*(constraintUV);  
  
 **this**.**\_baseboneConstraintUV**.set(Vec2f.*normalised*(constraintUV));  
 }  
  
 setBaseLocation(baseLocation: Vec2f): **void** {  
 **this**.**\_baseLocation**.set(baseLocation);  
 }  
  
 setBoneConnectionPoint(boneConnectionPoint: BoneConnectionPoint) { **this**.**\_boneConnectionPoint** = boneConnectionPoint; }  
  
 setChain(chain: FabrikBone2D[]): **void** {  
 **this**.**\_chain** = chain;  
 }  
  
 setColour(colour: Colour4f): **void** {  
 **for** (**let** aBone **of this**.**\_chain**) {  
 aBone.setColour(colour);  
 }  
 }  
  
 setConnectedBoneNumber(boneNumber: **number**): **void** {  
 **this**.**\_connectedBoneNumber** = boneNumber;  
 }  
  
 setConnectedChainNumber(chainNumber: **number**): **void** {  
 **this**.**\_connectedChainNumber** = chainNumber;  
 }  
  
 setFixedBaseMode(value: **boolean**): **void** {  
 **if** (!value && **this**.**\_connectedChainNumber** !== -1) {  
 **throw new** Error(**'This chain is connected to another chain so must remain in fixed base mode.'**);  
 }  
  
 **if** (**this**.**\_baseboneConstraintType** === BaseboneConstraintType2D.*GLOBAL\_ABSOLUTE* && !value) {  
 **throw new** Error(**'Cannot set a non-fixed base mode when the chain\'s constraint type is BaseBoneConstraintType2D.GLOBAL\_ABSOLUTE.'**);  
 }  
  
 **this**.**\_fixedBaseMode** = value;  
 }  
  
 setMaxIterationAttempts(maxIterations: **number**): **void** {  
 **if** (maxIterations < 1) {  
 **throw new** Error(**'The maximum number of attempts to solve this IK chain must be at least 1.'**);  
 }  
  
 **this**.**\_maxIterationAttempts** = maxIterations;  
 }  
  
 setMinIterationChange(minIterationChange: **number**): **void** {  
 **if** (minIterationChange < 0) {  
 **throw new** Error(**'The minimum iteration change value must be more than or equal to zero.'**);  
 }  
  
 **this**.**\_minIterationChange** = minIterationChange;  
 }  
  
 setName(name: **string**): **void** { **this**.**\_name** = name; }  
  
 setSolveDistanceThreshold(solveDistance: **number**): **void** {  
 **if** (solveDistance < 0.0) {  
 **throw new** Error(**'The solve distance threshold must be greater than or equal to zero.'**);  
 }  
  
 **this**.**\_solveDistanceThreshold** = solveDistance;  
 }  
  
 */\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@param*** *target Целевое местоположение.  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* **private** solveIK(target: Vec2f): **number** {  
 *// ---------- Прямой проход от эффектора к основанию -----------* **for** (**let** loop = **this**.**\_chain**.**length** - 1; loop >= 0; --loop) {  
 **const** thisBone = **this**.**\_chain**[loop];  
  
 **const** boneLength = thisBone.*getLength*();  
  
 **if** (loop !== **this**.**\_chain**.**length** - 1) {  
 **const** outerBone = **this**.**\_chain**[loop + 1];  
  
 **const** outerBoneOuterToInnerUV = outerBone.getDirectionUV().*negated*();  
  
 **const** thisBoneOuterToInnerUV = thisBone.getDirectionUV().*negated*();  
  
 **const** clockwiseConstraintDegs = outerBone.*getJoint*().*getClockwiseConstraintDegs*();  
 **const** antiClockwiseConstraintDegs = outerBone.*getJoint*().*getAnticlockwiseConstraintDegs*();  
  
  
 **let** constrainedUV;  
 **if** (**this**.**\_chain**[loop].*getJointConstraintCoordinateSystem*() == ConstraintCoordinateSystem.*LOCAL*) {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneOuterToInnerUV, outerBoneOuterToInnerUV, clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 } **else** {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneOuterToInnerUV, thisBone.*getGlobalConstraintUV*().*negated*(), clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 }  
  
  
 **const** newStartLocation = thisBone.*getEndLocation*().*plus*(constrainedUV.*times*(boneLength));  
  
 thisBone.setStartLocation(newStartLocation);  
  
 **if** (loop > 0) {  
 **this**.**\_chain**[loop - 1].setEndLocation(newStartLocation);  
 }  
 } **else** {  
 thisBone.setEndLocation(target);  
  
 **const** thisBoneOuterToInnerUV = thisBone.getDirectionUV().*negated*();  
  
 **let** constrainedUV;  
 **if** (loop > 0) {  
 **const** innerBoneOuterToInnerUV = **this**.**\_chain**[loop - 1].getDirectionUV().*negated*();  
  
 **const** clockwiseConstraintDegs = thisBone.*getJoint*().*getClockwiseConstraintDegs*();  
 **const** antiClockwiseConstraintDegs = thisBone.*getJoint*().*getAnticlockwiseConstraintDegs*();  
  
 **if** (thisBone.*getJoint*().*getConstraintCoordinateSystem*() == ConstraintCoordinateSystem.*LOCAL*) {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneOuterToInnerUV, innerBoneOuterToInnerUV, clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 }  
 **else** {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneOuterToInnerUV, thisBone.*getGlobalConstraintUV*().*negated*(), clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 }  
  
 } **else** {  
 **if** (thisBone.*getJointConstraintCoordinateSystem*() == ConstraintCoordinateSystem.*LOCAL*) {  
 constrainedUV = thisBoneOuterToInnerUV;  
 } **else** {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneOuterToInnerUV, thisBone.*getGlobalConstraintUV*().*negated*(), thisBone.*getClockwiseConstraintDegs*(), thisBone.getAnticlockwiseConstraintDegs());  
 }  
 }  
  
 **const** newStartLocation = thisBone.*getEndLocation*().*plus*(constrainedUV.*times*(boneLength));  
  
 thisBone.setStartLocation(newStartLocation);  
  
 **if** (loop > 0) {  
 **this**.**\_chain**[loop - 1].setEndLocation(newStartLocation);  
 }  
 }  
 }  
  
 *// ---------- Обратный проход от основания к эффектору -----------* **for** (**let** loop = 0; loop < **this**.**\_chain**.**length**; ++loop) {  
 **const** boneLength = **this**.**\_chain**[loop].*getLength*();  
  
 **const** thisBone = **this**.**\_chain**[loop];  
 **if** (loop !== 0) {  
 **const** previousBone = **this**.**\_chain**[loop - 1];  
  
 **const** thisBoneInnerToOuterUV = thisBone.getDirectionUV();  
 **const** prevBoneInnerToOuterUV = previousBone.getDirectionUV();  
  
 **const** clockwiseConstraintDegs = thisBone.*getJoint*().*getClockwiseConstraintDegs*();  
 **const** antiClockwiseConstraintDegs = thisBone.*getJoint*().*getAnticlockwiseConstraintDegs*();  
  
 **let** constrainedUV;  
 **if** (thisBone.*getJointConstraintCoordinateSystem*() == ConstraintCoordinateSystem.*LOCAL*) {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneInnerToOuterUV, prevBoneInnerToOuterUV, clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 } **else** {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneInnerToOuterUV, thisBone.*getGlobalConstraintUV*(), clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 }  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(constrainedUV.*times*(boneLength));  
  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (loop < **this**.**\_chain**.**length** - 1) {  
 **this**.**\_chain**[loop + 1].setStartLocation(newEndLocation);  
 }  
 } **else** {  
 **if** (**this**.**\_fixedBaseMode**) {  
 **this**.**\_chain**[0].setStartLocation(**this**.**\_baseLocation**);  
 } **else** {  
 **const** boneZeroUV = **this**.**\_chain**[0].getDirectionUV();  
 **const** boneZeroEndLocation = **this**.**\_chain**[0].*getEndLocation*();  
 **const** newBoneZeroStartLocation = boneZeroEndLocation.*minus*(boneZeroUV.times(boneLength));  
 **this**.**\_chain**[0].setStartLocation(newBoneZeroStartLocation);  
 }  
  
 **if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType2D.*NONE*) {  
 **const** thisBoneInnerToOuterUV = thisBone.getDirectionUV();  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(boneLength));  
  
 **this**.**\_chain**[0].setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 } **else** {  
 **const** thisBoneInnerToOuterUV = thisBone.getDirectionUV();  
  
 **const** clockwiseConstraintDegs = thisBone.*getJoint*().*getClockwiseConstraintDegs*();  
 **const** antiClockwiseConstraintDegs = thisBone.*getJoint*().*getAnticlockwiseConstraintDegs*();  
  
 **let** constrainedUV;  
 **if** (**this**.**\_baseboneConstraintType** === BaseboneConstraintType2D.*LOCAL\_ABSOLUTE*) {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneInnerToOuterUV, **this**.**\_baseboneRelativeConstraintUV**, clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
  
 } **else** {  
 constrainedUV = Vec2f.*getConstrainedUV*(thisBoneInnerToOuterUV, **this**.**\_baseboneConstraintUV**, clockwiseConstraintDegs, antiClockwiseConstraintDegs);  
 }  
 **const** newEndLocation = **this**.**\_chain**[loop].*getStartLocation*().*plus*(constrainedUV.*times*(boneLength));  
  
 **this**.**\_chain**[loop].setEndLocation(newEndLocation);  
  
 **if** (loop < **this**.**\_chain**.**length** - 1) {  
 **this**.**\_chain**[loop + 1].setStartLocation(newEndLocation);  
 }  
 }  
 }  
 }  
  
 **this**.**\_lastTargetLocation**.set(target);  
  
 **const** currentEffectorLocation = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 **return** Vec2f.*distanceBetween*(currentEffectorLocation, target);  
 }  
  
 */\*\*  
 \* Устанавливает должна ли использоваться встроенная целевая позиция.  
 \*  
 \** ***@param*** *value  
 \*/* setEmbeddedTargetMode(value: **boolean**): **void** { **this**.**\_useEmbeddedTarget** = value; }  
  
 **private** cloneChainVector(): FabrikBone2D[] {  
 **const** numBones = **this**.**\_chain**.**length**;  
  
 **const** clonedChain = [];  
  
 **for** (**let** loop = 0; loop < numBones; ++loop) {  
 **const** bone = **new** FabrikBone2D();  
 bone.set(**this**.**\_chain**[loop]);  
 clonedChain.push(bone);  
 }  
  
 **return** clonedChain;  
 }  
  
 updateChainLength(): **void** {  
 **this**.**\_chainLength** = 0.0;  
 **for** (**let** aBone **of this**.**\_chain**) {  
 **this**.**\_chainLength** += aBone.*getLength*();  
 }  
 }  
  
 updateEmbeddedTarget(newEmbeddedTarget: Vec2f): **void** {  
 **if** (**this**.**\_useEmbeddedTarget**) {  
 **this**.**\_embeddedTarget**.set(newEmbeddedTarget);  
 } **else** {  
 **throw new** Error(**'This chain does not have embedded targets enabled - enable with setEmbeddedTargetMode(true).'**);  
 }  
 }  
  
 */\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK для внутреннего положения.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* solveForEmbeddedTarget(): **number** {  
 **if** (**this**.**\_useEmbeddedTarget**) {  
 **return this**.solveForTarget(**this**.**\_embeddedTarget**);  
 } **else** {  
 **throw new** Error(**'This chain does not have embedded targets enabled - enable with setEmbeddedTargetMode(true).'**);  
 }  
 }  
  
 */\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@param*** *newTarget Целевое местоположение.  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* solveForTarget(newTarget: Vec2f): **number** {  
 **if** (**this**.**\_lastTargetLocation**.approximatelyEquals(newTarget, 0.001) && **this**.**\_lastBaseLocation**.approximatelyEquals(**this**.**\_baseLocation**, 0.001)) {  
 **return this**.**\_currentSolveDistance**;  
 }  
  
 **let** startingDistance;  
 **let** startingSolution = **null**;  
  
 **if** (**this**.**\_lastBaseLocation**.approximatelyEquals(**this**.**\_baseLocation**, 0.001)) {  
 startingDistance = Vec2f.*distanceBetween*(**this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*(), newTarget);  
 startingSolution = **this**.cloneChainVector();  
 } **else** {  
 startingDistance = Number.MAX\_VALUE;  
 }  
  
 **let** bestSolution: FabrikBone2D[] = [];  
  
 **let** bestSolveDistance = Number.MAX\_VALUE;  
 **let** lastPassSolveDistance = Number.MAX\_VALUE;  
  
 **let** solveDistance;  
 **for** (**let** loop = 0; loop < **this**.**\_maxIterationAttempts**; ++loop) {  
 solveDistance = **this**.solveIK(newTarget);  
  
 **if** (solveDistance < bestSolveDistance) {  
 bestSolveDistance = solveDistance;  
 bestSolution = **this**.cloneChainVector();  
  
 **if** (solveDistance <= **this**.**\_solveDistanceThreshold**) {  
 **break**;  
 }  
 } **else** {  
 **if** (Math.*abs*(solveDistance - lastPassSolveDistance) < **this**.**\_minIterationChange**) {  
 **break**;  
 }  
 }  
  
 lastPassSolveDistance = solveDistance;  
 }  
  
 **if** (bestSolveDistance < startingDistance) {  
 **this**.**\_currentSolveDistance** = bestSolveDistance;  
 **this**.**\_chain** = bestSolution;  
 } **else** {  
 **this**.**\_currentSolveDistance** = startingDistance;  
 **this**.**\_chain** = startingSolution;  
 }  
  
 **this**.**\_lastBaseLocation**.set(**this**.**\_baseLocation**);  
 **this**.**\_lastTargetLocation**.set(newTarget);  
  
 **return this**.**\_currentSolveDistance**;  
 }  
  
 getBaseboneRelativeConstraintUV(): Vec2f { **return this**.**\_baseboneRelativeConstraintUV**; }  
  
 setBaseboneRelativeConstraintUV(constraintUV: Vec2f): **void** { **this**.**\_baseboneRelativeConstraintUV**.set(constraintUV); }  
  
 getMaxIterationAttempts(): **number** {  
 **return this**.**\_maxIterationAttempts**;  
 }  
  
 getMinIterationChange(): **number** {  
 **return this**.**\_minIterationChange**;  
 }  
  
 getSolveDistanceThreshold(): **number** {  
 **return this**.**\_solveDistanceThreshold**;  
 }  
}

### FabrikChain3D

**import** {FabrikBone3D} **from './FabrikBone3D'**;  
**import** {FabrikJoint3D} **from './FabrikJoint3D'**;  
**import** {FabrikStructure3D} **from './FabrikStructure3D'**;  
**import** {JointType} **from './JointType'**;  
**import** {Vec3f} **from '../utils/Vec3f'**;  
**import** {BaseboneConstraintType3D} **from './BaseboneConstraintType3D'**;  
**import** {Utils} **from '../utils/Utils'**;  
**import** {Colour4f} **from '../utils/Colour4f'**;  
**import** {Mat3f} **from '../utils/Mat3f'**;  
  
*/\*\*  
 \* 3D цепь костей, решающая задачу инверсной кинематики с помощью алгоритма FABRIK.  
 \*/***export class** FabrikChain3D {  
 *// ---------- Private Properties ----------  
 /\*\*  
 \* Массив костей цепи.  
 \*/* **private \_chain**: FabrikBone3D[] = [];  
  
 */\*\*  
 \* Название цепи.  
 \* <p>  
 \* Необязательное к использованию поле, помогающее идентифицировать цепь.  
 \*/* **private \_name**: **string**;  
  
 */\*\*  
 \* Порог, достигнув которого мы считаем задачу инверсной кинематики решенной.  
 \* <p>  
 \* При решении задачи ИК, когда расстояние между текущим положением и желаемым положением становится меньше или равно  
 \* значению порога, мы прерываем дальнейшие итерации и считаем задачу решенной.  
 \* <p>  
 \* Значение по умолчанию 0.01.  
 \* <p>  
 \* Минимально допустимое значение - 0, но стоит помнить о погрешности вычисления и не задавать порог равным нулю, так  
 \* как это может привести к бесконечным бесполезным попыткам.  
 \* <p>  
 \* Хотя данное поле и самый важный критерий останова для ИК-алгоритма, оно работает в комбинации с  
 \* {****@link*** *#\_maxIterationAttempts} и {****@link*** *\_minIterationChange}. Эти вспомогательные свойства помогают пережить ситуации,  
 \* когда длины цепи не хватает для достижения цели или когда суставы цепи сильно ограничены.  
 \*/* **private \_solveDistanceThreshold** = 0.01;  
  
 */\*\*  
 \* Максимальное количество итераций алгоритма, применяемых при решении задачи ИК.  
 \*  
 \** ***@default*** *20  
 \*/* **private \_maxIterationAttempts** = 20;  
  
 */\*\*  
 \* Минимальное значения для дельты рассотояний между итерациями алгоритма.  
 \*  
 \** ***@default*** *0.01  
 \*/* **private \_minIterationChange** = 0.01;  
  
 */\*\*  
 \* Длина цепи.  
 \*/* **private \_chainLength**: **number**;  
  
 */\*\*  
 \* Начальная позиция первой кости в цепи.  
 \*  
 \** ***@default:*** *Vec3f(0, 0)  
 \*/* **private \_fixedBaseLocation** = **new** Vec3f();  
  
 */\*\*  
 \* Зафиксирована ли начальная позиция цепи.  
 \* Если нет - то она может перемещаться.  
 \*  
 \** ***@default*** *true  
 \*/* **private \_fixedBaseMode** = **true**;  
  
 */\*\*  
 \* Тип сустава первой кости в цепи.  
 \*/* **private \_baseboneConstraintType**: BaseboneConstraintType3D = BaseboneConstraintType3D.*NONE*;  
  
 */\*\*  
 \* Направление относительно которого мы ограничиваем первую кость цепи.  
 \*/* **private \_baseboneConstraintUV** = **new** Vec3f();  
  
 **private \_baseboneRelativeConstraintUV**: Vec3f = **new** Vec3f();  
  
 **private \_baseboneRelativeReferenceConstraintUV**: Vec3f = **new** Vec3f();  
  
 */\*\*  
 \* Последняя целевая позиция, относительно которой решалась задача ИК.  
 \*  
 \** ***@default*** *Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE)  
 \*/* **private \_lastTargetLocation**: Vec3f = **new** Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE);  
  
 */\*\*  
 \* Предыдущая начальная позиция базовой (первой в цепи) кости.  
 \* <p>  
 \* Данное значение хранится для того, чтобы сравнивать изменилось ли  
 \* начальное положение базовой кости в процессе решения.  
 \*  
 \** ***@default*** *Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE)  
 \*/* **private \_lastBaseLocation**: Vec3f = **new** Vec3f(Number.MAX\_VALUE, Number.MAX\_VALUE, Number.MAX\_VALUE);  
  
 */\*\*  
 \* Текущее расстояние между эффектором(концом последней точки в цепи) и целевой позицией.  
 \*/* **private \_currentSolveDistance**: **number** = Number.MAX\_VALUE;  
  
 */\*\*  
 \* Номер цепи, к которой эта цепь подключена в FabrikStructure3D.  
 \* <p>  
 \* Если значение -1, значит данная цепь ни с чем не связана.  
 \*  
 \** ***@default*** *-1  
 \*/* **private \_connectedChainNumber**: **number** = -1;  
  
 */\*\*  
 \* Номер кости, с которой связана эта цепь, если она вообще связана с другой цепью.  
 \* <p>  
 \* Если значение -1, значит данная цепь ни с чем не связана.  
 \*  
 \** ***@default*** *-1  
 \*/* **private \_connectedBoneNumber**: **number** = -1;  
  
 */\*\*  
 \* Встроенное целевое расположение,которое можно использовать для решения ИК.  
 \* <p>  
 \* Встроенные целевые местоположения позволяют решать ИК структуры для нескольких целей (по одной на цепочку в структуре)  
 \* вместо того, чтобы все цепочки решались для одной и той же цели. Чтобы использовать встроенные цели, флаг \_useEmbeddedTargets  
 \* должно быть true(что не является значением по умолчанию) - этот флаг можно установить с помощью вызова setEmbeddedTargetMode (true).  
 \*/* **private \_embeddedTarget**: Vec3f = **new** Vec3f();  
  
 */\*\*  
 \* \_useEmbeddedTarget Следует ли использовать местоположение \_embeddedTarget при решении этой цепочки.  
 \* <p>  
 \* Этот флаг можно переключить, вызвав setEmbeddedTargetMode (true) в цепочке.  
 \*  
 \** ***@default*** *false  
 \** ***@see*** *{****@link*** *setEmbeddedTargetMode(boolean) }  
 \*/* **private \_useEmbeddedTarget**: **boolean** = **false**;  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \*  
 \* Принимает три вариации параметров:  
 \*  
 \* Без параметров. Всем полям проставляются значения по умолчанию.  
 \*  
 \* Один параметр-строка.  
 \*  
 \** ***@param*** *source Имя данной цепи.  
 \*  
 \* Конструктор копирования.  
 \*  
 \** ***@param*** *source Цепь для копирования.  
 \*/* **constructor**(source?: **string** | FabrikChain3D) {  
 **if** (!source) {  
 **return**;  
 }  
 **if** (**typeof** source === **'string'**) {  
 **this**.**\_name** = source;  
 } **else** {  
 **this**.**\_chain** = source.cloneIkChain();  
  
 **this**.**\_fixedBaseLocation**.set(source.*getBaseLocation*());  
 **this**.**\_lastTargetLocation**.set(source.**\_lastTargetLocation**);  
 **this**.**\_lastBaseLocation**.set(source.**\_lastBaseLocation**);  
 **this**.**\_embeddedTarget**.set(source.**\_embeddedTarget**);  
  
 **if** (source.**\_baseboneConstraintType** != BaseboneConstraintType3D.*NONE*) {  
 **this**.**\_baseboneConstraintUV**.set(source.**\_baseboneConstraintUV**);  
 **this**.**\_baseboneRelativeConstraintUV**.set(source.**\_baseboneRelativeConstraintUV**);  
 }  
  
 **this**.**\_chainLength** = source.**\_chainLength**;  
 **this**.**\_currentSolveDistance** = source.**\_currentSolveDistance**;  
 **this**.**\_connectedChainNumber** = source.**\_connectedChainNumber**;  
 **this**.**\_connectedBoneNumber** = source.**\_connectedBoneNumber**;  
 **this**.**\_baseboneConstraintType** = source.**\_baseboneConstraintType**;  
 **this**.**\_name** = source.**\_name**;  
 **this**.**\_useEmbeddedTarget** = source.**\_useEmbeddedTarget**;  
 }  
 }  
  
 *// ---------- Public Methods ------------  
  
 /\*\*  
 \* Добавляет кость в конец цепи.  
 \*  
 \** ***@param*** *bone Кость.  
 \*/* addBone(bone: FabrikBone3D): **void** {  
 **this**.**\_chain**.push(bone);  
  
 **if** (**this**.**\_chain**.**length** == 1) {  
 **this**.**\_fixedBaseLocation**.set(bone.*getStartLocation*());  
  
 **this**.**\_baseboneConstraintUV** = bone.getDirectionUV();  
 }  
  
 **this**.updateChainLength();  
 }  
  
 */\*\*\*  
 \* Добавляет последовательную кость в конец этой цепи IK с учетом вектора направления и длины новой кости.  
 \* <p>  
 \* Добавленная кость не имеет каких-либо ограничений.  
 \* <p>  
 \* Этот метод можно использовать только тогда, когда цепь IK содержит базовую кость, так как без нее у нас нет  
 \* начального местоположения для этой кости (т. е. конечного местоположения предыдущей).  
 \* <p>  
 \* Если этот метод выполняется в цепочке, которая не содержит основной кости, то возбуждается ошибка.  
 \* <p>  
 \* Если вектор направления или длина равны нулю, то возбуждается ошибка.  
 \*  
 \** ***@param*** *directionUV Первоначальное направление новой кости.  
 \** ***@param*** *length Длина новой кости.  
 \** ***@param*** *colour Цвет для отрисовки новой кости.  
 \** ***@param*** *name Имя новой кости.  
 \*/* addConsecutiveBone(directionUV: Vec3f, length: **number**, colour?: Colour4f, name?: **string**): **void** {  
 Utils.*validateDirectionUV*(directionUV);  
  
 Utils.*validateLength*(length);  
  
 **if** (!(**this**.**\_chain**.**length** === 0)) {  
 **const** prevBoneEnd: Vec3f = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 **this**.addBone(**new** FabrikBone3D(prevBoneEnd, directionUV.normalised(), length, colour, name));  
 } **else** {  
 **throw new** Error(**'You cannot add the basebone as a consecutive bone as it does not provide a start location. Use the addBone() method instead.'**);  
 }  
 }  
  
 */\*\*  
 \* Добавляет предварительно созданную последовательную кость к концу этой цепочки IK.  
 \* <p>  
 \* Этот метод можно использовать только тогда, когда цепь IK содержит базовую кость, так как без нее у нас нет  
 \* начального местоположения для этой кости (т. е. конечного местоположения предыдущей).  
 \* <p>  
 \* Если этот метод выполняется в цепочке, которая не содержит основной кости,  
 \* или кость, которую пытаются добавить имеет нулевую длину то возбуждается ошибка.  
 \*  
 \** ***@param*** *bone Кость.  
 \*/* addConsecutiveCreatedBone(bone: FabrikBone3D): **void** {  
 **const** dir: Vec3f = bone.getDirectionUV();  
 Utils.*validateDirectionUV*(dir);  
  
 **const** len = bone.*getLength*();  
 Utils.*validateLength*(len);  
  
 **if** (!(**this**.**\_chain**.**length** === 0)) {  
 **const** prevBoneEnd: Vec3f = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 bone.setStartLocation(prevBoneEnd);  
 bone.setEndLocation(prevBoneEnd.*plus*(dir.times(len)));  
  
 **this**.addBone(bone);  
 } **else** {  
 **throw new** Error(**'You cannot add the base bone to a chain using this method as it does not provide a start location.'**);  
 }  
 }  
  
 */\*\*  
 \* Добавляет последовательную шарнирную кость к концу этой цепи IK.  
 \* <p>  
 \* Тип шарнира может быть глобальным, ось вращения которого указана в мировом пространстве, или  
 \* локальным, ось вращения которого расположена относительно предыдущей кости в цепи.  
 \* <p>  
 \* Если этот метод выполняется в цепочке, которая не содержит основной кости, возникает ошибка.  
 \* Если направление кости или ось ограничения шарнира нули, возникает ошибка.  
 \* Если запрошенный тип соединения не является LOCAL\_HINGE или GLOBAL\_HINGE, возникает ошибка.  
 \*  
 \** ***@param*** *directionUV Начальное направление новой кости.  
 \** ***@param*** *length Длина новой кости.  
 \** ***@param*** *jointType Тип сустава новой кости.  
 \** ***@param*** *hingeRotationAxis Ось вращения шарнира.  
 \** ***@param*** *clockwiseDegs Угол-ограничение по часовой стрелке в градусах.  
 \** ***@param*** *anticlockwiseDegs Угол-ограничение против часовой стрелки в градусах.  
 \** ***@param*** *hingeReferenceAxis Ось относительно которой ведется отсчет угла поворота шарнира.  
 \** ***@param*** *colour Цвет новой кости.  
 \*/* addConsecutiveHingedBone(  
 directionUV: Vec3f,  
 length: **number**,  
 jointType: JointType,  
 hingeRotationAxis: Vec3f,  
 clockwiseDegs: **number** = 180,  
 anticlockwiseDegs: **number** = 180,  
 hingeReferenceAxis: Vec3f = Vec3f.*genPerpendicularVectorQuick*(hingeRotationAxis),  
 colour: Colour4f = **new** Colour4f(),  
 ): **void** {  
 Utils.*validateDirectionUV*(directionUV);  
 Utils.*validateDirectionUV*(hingeRotationAxis);  
 Utils.*validateLength*(length);  
  
 **if** (**this**.**\_chain**.**length** === 0) {  
 **throw new** Error(**'You must add a basebone before adding a consectutive bone.'**);  
 }  
  
 directionUV.normalise();  
 hingeRotationAxis.normalise();  
  
 **const** prevBoneEnd: Vec3f = **this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
  
 **const** bone: FabrikBone3D = **new** FabrikBone3D(prevBoneEnd, directionUV, length, colour);  
  
 **const** joint: FabrikJoint3D = **new** FabrikJoint3D();  
 **switch** (jointType) {  
 **case** JointType.*GLOBAL\_HINGE*:  
 joint.setHingeJoint(JointType.*GLOBAL\_HINGE*, hingeRotationAxis, clockwiseDegs, anticlockwiseDegs, hingeReferenceAxis);  
 **break**;  
 **case** JointType.*LOCAL\_HINGE*:  
 joint.setHingeJoint(JointType.*LOCAL\_HINGE*, hingeRotationAxis, clockwiseDegs, anticlockwiseDegs, hingeReferenceAxis);  
 **break**;  
 **default**:  
 **throw new** Error(**'Hinge joint types may be only JointType.GLOBAL\_HINGE or JointType.LOCAL\_HINGE.'**);  
 }  
  
 bone.setJoint(joint);  
  
 **this**.addBone(bone);  
 }  
  
 */\*\*  
 \* Добавляет последовательный шаровой сустав, ограничивающий кость, к концу этой цепи IK.  
 \* <p>  
 \* Этот метод можно использовать только тогда, когда цепь IK содержит базовую кость, так как без нее у нас нет  
 \* начального местоположения для этой кости (т. е. конечного местоположения предыдущей).  
 \* <p>  
 \* Если этот метод выполняется в цепочке, которая не содержит основной кости,  
 \* или кость, которую пытаются добавить имеет нулевую длину то возбуждается ошибка.  
 \*  
 \** ***@param*** *boneDirectionUV Начальное направление единичного вектора новой кости.  
 \** ***@param*** *boneLength Длина новой кости.  
 \** ***@param*** *constraintAngleDegs Угол ограничение.  
 \** ***@param*** *colour Цвет для отрисовки новой кости.  
 \*/* addConsecutiveRotorConstrainedBone(  
 boneDirectionUV: Vec3f,  
 boneLength: **number**,  
 constraintAngleDegs: **number** = 180,  
 colour: Colour4f = **new** Colour4f(),  
 ): **void** {  
 Utils.*validateDirectionUV*(boneDirectionUV);  
 Utils.*validateLength*(boneLength);  
 **if** (**this**.**\_chain**.**length** === 0) {  
 **throw new** Error(**'Add a basebone before attempting to add consectuive bones.'**);  
 }  
  
 **const** bone: FabrikBone3D = **new** FabrikBone3D(**this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*(), boneDirectionUV.normalise(), boneLength, colour);  
 bone.setBallJointConstraintDegs(constraintAngleDegs);  
 **this**.addBone(bone);  
 }  
  
 */\*\*  
 \* Возвращает относительный единичный вектор основной кости этой цепочки.  
 \*  
 \** ***@return*** *\*/  
 getBaseboneRelativeConstraintUV* = (): Vec3f => **this**.**\_baseboneRelativeConstraintUV**;  
  
 */\*\*  
 \* Возвращает тип сустава базовой кости.  
 \*  
 \** ***@return*** *\*/  
 getBaseboneConstraintType* = (): BaseboneConstraintType3D => **this**.**\_baseboneConstraintType**;  
  
 */\*\*  
 \* Возвращает ограничение направления основной кости.  
 \* <p>  
 \* Если базовая кость не ограничена, то возникает ошибка. Если вы хотите проверить, ограничена ли  
 \* базовая кость этой цепочки ИК вы можете использовать метод {****@link*** *#getBaseboneConstraintType()}.  
 \*  
 \** ***@return*** *\*/* getBaseboneConstraintUV(): Vec3f {  
 **if** (**this**.**\_baseboneConstraintType** != BaseboneConstraintType3D.*NONE*) {  
 **return this**.**\_baseboneConstraintUV**;  
 } **else** {  
 **throw new** Error(**'Cannot return the basebone constraint when the basebone constraint type is NONE.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает базовое расположение цепочки IK.  
 \* <p>  
 \* Независимо от того, сколько костей содержится в цепи, базовое местоположение всегда является начальным местоположением  
 \* первой кости в цепи.  
 \*  
 \** ***@return*** *\*/  
 getBaseLocation* = (): Vec3f => **this**.**\_chain**[0].*getStartLocation*();  
  
 */\*\*  
 \* Возвращает кость цепи по индексу.  
 \*  
 \** ***@param*** *boneNumber Индекс кости.  
 \** ***@return*** *Кость.  
 \*/  
 getBone* = (boneNumber: **number**): FabrikBone3D => **this**.**\_chain**[boneNumber];  
  
 */\*\*  
 \* Возвращает массив костей цепи.  
 \*  
 \** ***@return*** *\*/* getChain(): FabrikBone3D[] {  
 **return this**.**\_chain**;  
 }  
  
 */\*\*  
 \* Возвращает длину цепи.  
 \*  
 \** ***@return*** *\*/  
 getChainLength* = (): **number** => **this**.**\_chainLength**;  
  
 */\*\*  
 \* Возвращает индекс кости из другой цепи, к которой присоединена наша цепь.  
 \* <p>  
 \* Возвращает -1 если мы ни к кому не присоединены.  
 \*  
 \** ***@return*** *\*/  
 getConnectedBoneNumber* = (): **number** => **this**.**\_connectedBoneNumber**;  
  
 */\*\*  
 \* Возвращает индекс цепи в структуре, к которой присоединена наша цепь.  
 \* <p>  
 \* Возвращает -1 если мы ни к кому не присоединены.  
 \*  
 \** ***@return*** *\*/  
 getConnectedChainNumber* = (): **number** => **this**.**\_connectedChainNumber**;  
  
 */\*\*  
 \* Возвращает эффектор.  
 \* <p>  
 \* Независимо от того, сколько костей содержится в цепи, эффектор - это всегда конец  
 \* последней кости в цепи.  
 \*  
 \** ***@return*** *\*/* getEffectorLocation(): Vec3f {  
 **return this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*();  
 }  
  
 */\*\*  
 \* Возвращает должна ли цепь использовать встроенное целевое положение.  
 \*  
 \** ***@return*** *\*/* getEmbeddedTargetMode(): **boolean** {  
 **return this**.**\_useEmbeddedTarget**;  
 }  
  
 */\*\*  
 \* Возврат внедренного целевого местоположения.  
 \*  
 \** ***@return*** *\*/  
 getEmbeddedTarget* = (): Vec3f => **this**.**\_embeddedTarget**;  
  
 */\*\*  
 \* Возвращает цель последней попытки решения.  
 \* <p>  
 \* Целевое местоположение и местоположение эффектора не обязательно находятся в одном и том же месте.  
 \*  
 \** ***@return*** *\*/  
 getLastTargetLocation* = (): Vec3f => **this**.**\_lastTargetLocation**;  
  
 */\*\*  
 \* Возвращает название этой ИК цепи.  
 \*  
 \** ***@return*** *\*/  
 getName* = (): **string** => **this**.**\_name**;  
  
 */\*\*  
 \* Возвращает количество костей в цепи.  
 \*  
 \** ***@return*** *\*/  
 getNumBones* = (): **number** => **this**.**\_chain**.**length**;  
  
 */\*\*  
 \* Удаляет кость из этой цепи IK по ее индексу в цепи.  
 \* <p>  
 \* Если номер кости, которую необходимо удалить, не существует в цепочке, то возникает ошибка.  
 \*  
 \** ***@param*** *boneNumber Индекс кости в цепи.  
 \*/* removeBone(boneNumber: **number**): **void** {  
 **if** (boneNumber < **this**.**\_chain**.**length**) {  
 **this**.**\_chain**.splice(boneNumber, 1);  
 **this**.updateChainLength();  
 } **else** {  
 **throw new** Error(**'Bone '** + boneNumber + **' does not exist to be removed from the chain. Bones are zero indexed.'**);  
 }  
 }  
  
 \_setBaseboneRelativeConstraintUV(constraintUV: Vec3f): **void** {  
 **this**.**\_baseboneRelativeConstraintUV** = constraintUV;  
 }  
  
 \_setBaseboneRelativeReferenceConstraintUV(constraintUV: Vec3f): **void** {  
 **this**.**\_baseboneRelativeReferenceConstraintUV** = constraintUV;  
 }  
  
 *getBaseboneRelativeReferenceConstraintUV* = (): Vec3f => **this**.**\_baseboneRelativeReferenceConstraintUV**;  
  
 */\*\*  
 \* Устанавливает должна ли использоваться встроенная целевая позиция.  
 \*  
 \** ***@param*** *value  
 \*/* setEmbeddedTargetMode(value: **boolean**): **void** {  
 **this**.**\_useEmbeddedTarget** = value;  
 }  
  
 setRotorBaseboneConstraint(rotorType: BaseboneConstraintType3D, constraintAxis: Vec3f, angleDegs: **number**): **void** {  
 **if** (**this**.**\_chain**.**length** === 0) {  
 **throw new** Error(**'Chain must contain a basebone before we can specify the basebone constraint type.'**);  
 }  
 **if** (constraintAxis.*length*() <= 0.0) {  
 **throw new** Error(**'Constraint axis cannot be zero.'**);  
 }  
 **if** (angleDegs < 0.0) {  
 angleDegs = 0.0;  
 }  
 **if** (angleDegs > 180.0) {  
 angleDegs = 180.0;  
 }  
 **if** (!(rotorType == BaseboneConstraintType3D.*GLOBAL\_ROTOR* || rotorType == BaseboneConstraintType3D.*LOCAL\_ROTOR*)) {  
 **throw new** Error(**'The only valid rotor types for this method are GLOBAL\_ROTOR and LOCAL\_ROTOR.'**);  
 }  
  
 **this**.**\_baseboneConstraintType** = rotorType;  
 **this**.**\_baseboneConstraintUV** = constraintAxis.normalised();  
 **this**.**\_baseboneRelativeConstraintUV**.set(**this**.**\_baseboneConstraintUV**);  
 **this**.*getBone*(0).*getJoint*().setAsBallJoint(angleDegs);  
 }  
  
 setHingeBaseboneConstraint(hingeType: BaseboneConstraintType3D, hingeRotationAxis: Vec3f, cwConstraintDegs: **number**, acwConstraintDegs: **number**, hingeReferenceAxis: Vec3f): **void** {  
 **if** (**this**.**\_chain**.**length** === 0) {  
 **throw new** Error(**'Chain must contain a basebone before we can specify the basebone constraint type.'**);  
 }  
 **if** (hingeRotationAxis.*length*() <= 0.0) {  
 **throw new** Error(**'Hinge rotation axis cannot be zero.'**);  
 }  
 **if** (hingeReferenceAxis.*length*() <= 0.0) {  
 **throw new** Error(**'Hinge reference axis cannot be zero.'**);  
 }  
 **if** (!(Vec3f.*perpendicular*(hingeRotationAxis, hingeReferenceAxis))) {  
 **throw new** Error(**'The hinge reference axis must be in the plane of the hinge rotation axis, that is, they must be perpendicular.'**);  
 }  
 **if** (!(hingeType == BaseboneConstraintType3D.*GLOBAL\_HINGE* || hingeType == BaseboneConstraintType3D.*LOCAL\_HINGE*)) {  
 **throw new** Error(**'The only valid hinge types for this method are GLOBAL\_HINGE and LOCAL\_HINGE.'**);  
 }  
  
 **this**.**\_baseboneConstraintType** = hingeType;  
 **this**.**\_baseboneConstraintUV**.set(hingeRotationAxis.normalised());  
  
 **const** hinge: FabrikJoint3D = **new** FabrikJoint3D();  
  
 **if** (hingeType == BaseboneConstraintType3D.*GLOBAL\_HINGE*) {  
 hinge.setHingeJoint(JointType.*GLOBAL\_HINGE*, hingeRotationAxis, cwConstraintDegs, acwConstraintDegs, hingeReferenceAxis);  
 } **else** {  
 hinge.setHingeJoint(JointType.*LOCAL\_HINGE*, hingeRotationAxis, cwConstraintDegs, acwConstraintDegs, hingeReferenceAxis);  
 }  
 **this**.*getBone*(0).setJoint(hinge);  
 }  
  
 setBaseboneConstraintUV(constraintUV: Vec3f): **void** {  
 **if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*NONE*) {  
 **throw new** Error(**'Specify the basebone constraint type with setBaseboneConstraintTypeCannot specify a basebone constraint when the current constraint type is BaseboneConstraint.NONE.'**);  
 }  
  
 Utils.*validateDirectionUV*(constraintUV);  
  
 constraintUV.normalise();  
 **this**.**\_baseboneConstraintUV**.set(constraintUV);  
 }  
  
 setBaseLocation(baseLocation: Vec3f): **void** {  
 **this**.**\_fixedBaseLocation**.set(baseLocation);  
 }  
  
 */\*\*  
 \* Присоединяет цепь к кости из другой цепи.  
 \* <p>  
 \* Чтобы соединить эту цепь с другой цепью, обе цепи должны существовать в одной и той же структуре.  
 \* <p>  
 \* Если структура не содержит указанную цепь или кость, то возникает ошибка.  
 \*  
 \** ***@param*** *structure Структура, в которой произойдёт соединение.  
 \** ***@param*** *chainNumber Индекс цепи для присоединения.  
 \** ***@param*** *boneNumber Индекс кости к которой мы присоединяемся.  
 \*/* connectToStructure(structure: FabrikStructure3D, chainNumber: **number**, boneNumber: **number**): **void** {  
 **const** nu\_chains = structure.getNumChains();  
 **if** (chainNumber > nu\_chains) {  
 **throw new** Error(**'Structure does not contain a chain '** + chainNumber + **' - it has '** + nu\_chains + **' chains.'**);  
 }  
  
 **const** numBones = structure.getChain(chainNumber).*getNumBones*();  
 **if** (boneNumber > numBones) {  
 **throw new** Error(**'Chain does not contain a bone '** + boneNumber + **' - it has '** + numBones + **' bones.'**);  
 }  
  
 **this**.**\_connectedChainNumber** = chainNumber;  
 **this**.**\_connectedBoneNumber** = boneNumber;  
 }  
  
 setFixedBaseMode(value: **boolean**): **void** {  
 **if** (!value && **this**.**\_connectedChainNumber** != -1) {  
 **throw new** Error(**'This chain is connected to another chain so must remain in fixed base mode.'**);  
 }  
  
 **if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*GLOBAL\_ROTOR* && !value) {  
 **throw new** Error(**'Cannot set a non-fixed base mode when the chain\'s constraint type is BaseboneConstraintType3D.GLOBAL\_ABSOLUTE\_ROTOR.'**);  
 }  
  
 **this**.**\_fixedBaseMode** = value;  
 }  
  
 setMaxIterationAttempts(maxIterations: **number**): **void** {  
 **if** (maxIterations < 1) {  
 **throw new** Error(**'The maximum number of attempts to solve this IK chain must be at least 1.'**);  
 }  
  
 **this**.**\_maxIterationAttempts** = maxIterations;  
 }  
  
 setMinIterationChange(minIterationChange: **number**): **void** {  
 **if** (minIterationChange < 0.0) {  
 **throw new** Error(**'The minimum iteration change value must be more than or equal to zero.'**);  
 }  
  
 **this**.**\_minIterationChange** = minIterationChange;  
 }  
  
 setName(name: **string**): **void** {  
 **this**.**\_name** = name;  
 }  
  
 setSolveDistanceThreshold(solveDistance: **number**): **void** {  
 **if** (solveDistance < 0.0) {  
 **throw new** Error(**'The solve distance threshold must be greater than or equal to zero.'**);  
 }  
  
 **this**.**\_solveDistanceThreshold** = solveDistance;  
 }  
  
 setColour(colour: Colour4f): **void** {  
 **for** (**let** aBone **of this**.**\_chain**) {  
 aBone.setColour(colour);  
 }  
 }  
  
 */\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK для внутреннего положения.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* solveForEmbeddedTarget(): **number** {  
 **if** (**this**.**\_useEmbeddedTarget**) {  
 **return this**.solveForTarget(**this**.**\_embeddedTarget**);  
 } **else** {  
 **throw new** Error(**'This chain does not have embedded targets enabled - enable with setEmbeddedTargetMode(true).'**);  
 }  
 }  
  
 */\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@param*** *newTarget Целевое местоположение.  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* solveForTarget(newTarget: Vec3f): **number** {  
 **if** (**this**.**\_lastTargetLocation**.approximatelyEquals(newTarget, 0.00001) &&  
 (!**this**.**\_fixedBaseMode** && **this**.**\_lastBaseLocation**.approximatelyEquals(**this**.*getBaseLocation*(), 0.00001)) &&  
 (**this**.**\_fixedBaseMode** && **this**.**\_fixedBaseLocation**.approximatelyEquals(**this**.*getBaseLocation*(), 0.00001))  
 ) {  
 **return this**.**\_currentSolveDistance**;  
 }  
  
 **let** bestSolution: FabrikBone3D[] = [];  
  
 **let** bestSolveDistance = Number.MAX\_VALUE;  
  
 **let** lastPassSolveDistance = Number.MAX\_VALUE;  
  
 **let** solveDistance;  
 **for** (**let** loop = 0; loop < **this**.**\_maxIterationAttempts**; ++loop) {  
 solveDistance = **this**.solveIK(newTarget);  
  
 **if** (solveDistance < bestSolveDistance) {  
 bestSolveDistance = solveDistance;  
 bestSolution = **this**.cloneIkChain();  
  
 **if** (solveDistance <= **this**.**\_solveDistanceThreshold**) {  
 **break**;  
 }  
 } **else** {  
 **if** (Math.*abs*(solveDistance - lastPassSolveDistance) < **this**.**\_minIterationChange**) {  
 **break**;  
 }  
 }  
  
 lastPassSolveDistance = solveDistance;  
  
 }  
 **this**.**\_currentSolveDistance** = bestSolveDistance;  
 **this**.**\_chain** = bestSolution;  
  
 **this**.**\_lastBaseLocation**.set(**this**.*getBaseLocation*());  
 **this**.**\_lastTargetLocation**.set(newTarget);  
  
 **return this**.**\_currentSolveDistance**;  
 }  
  
*// ---------- Private Methods ----------  
  
 /\*\*  
 \* Решает задачу ИК с помощью алгоритма FABRIK.  
 \* <p>  
 \* Если в цепи еще нет костей - возникнет ошибка.  
 \*  
 \** ***@param*** *target Целевое местоположение.  
 \** ***@return*** *Наименьшее расстояние между новым положением эффектора и целевым местоположением, которого удалось достичь.  
 \*/* **private** solveIK(target: Vec3f): **number** {  
 **if** (**this**.**\_chain**.**length** === 0) {  
 **throw new** Error(**'It makes no sense to solve an IK chain with zero bones.'**);  
 }  
  
 *// ---------- ПРЯМОЙ ПРОХОД: от эффектора к базовой кости -----------* **for** (**let** loop = **this**.**\_chain**.**length** - 1; loop >= 0; --loop) {  
 **const** thisBone: FabrikBone3D = **this**.**\_chain**[loop];  
 **const** thisBoneLength: **number** = thisBone.*getLength*();  
 **const** thisBoneJoint: FabrikJoint3D = thisBone.*getJoint*();  
 **const** thisBoneJointType: JointType = thisBone.*getJointType*();  
  
 **if** (loop != **this**.**\_chain**.**length** - 1) {  
 **const** outerBoneOuterToInnerUV: Vec3f = **this**.**\_chain**[loop + 1].getDirectionUV().*negated*();  
  
 **let** thisBoneOuterToInnerUV: Vec3f = thisBone.getDirectionUV().*negated*();  
  
 **if** (thisBoneJointType == JointType.*BALL*) {  
 **const** angleBetweenDegs = Vec3f.*getAngleBetweenDegs*(outerBoneOuterToInnerUV, thisBoneOuterToInnerUV);  
 **const** constraintAngleDegs = thisBoneJoint.getBallJointConstraintDegs();  
 **if** (angleBetweenDegs > constraintAngleDegs) {  
 thisBoneOuterToInnerUV = Vec3f.*getAngleLimitedUnitVectorDegs*(thisBoneOuterToInnerUV, outerBoneOuterToInnerUV, constraintAngleDegs);  
 }  
 } **else if** (thisBoneJointType == JointType.*GLOBAL\_HINGE*) {  
 thisBoneOuterToInnerUV = thisBoneOuterToInnerUV.projectOntoPlane(thisBoneJoint.getHingeRotationAxis());  
 } **else if** (thisBoneJointType == JointType.*LOCAL\_HINGE*) {  
 **let** m;  
 **let** relativeHingeRotationAxis;  
 **if** (loop > 0) {  
 m = Mat3f.*createRotationMatrix*(**this**.**\_chain**[loop - 1].getDirectionUV());  
 relativeHingeRotationAxis = m.*times*(thisBoneJoint.getHingeRotationAxis()).normalise();  
 } **else** {  
 relativeHingeRotationAxis = **this**.**\_baseboneRelativeConstraintUV**;  
 }  
  
 thisBoneOuterToInnerUV = thisBoneOuterToInnerUV.projectOntoPlane(relativeHingeRotationAxis);  
 }  
  
 **let** newStartLocation = thisBone.*getEndLocation*().*plus*(thisBoneOuterToInnerUV.times(thisBoneLength));  
  
 thisBone.setStartLocation(newStartLocation);  
  
 **if** (loop > 0) {  
 **this**.**\_chain**[loop - 1].setEndLocation(newStartLocation);  
 }  
 } **else** {  
  
 thisBone.setEndLocation(target);  
  
 **let** thisBoneOuterToInnerUV = thisBone.getDirectionUV().*negated*();  
  
 **switch** (thisBoneJointType) {  
 **case** JointType.*BALL*:  
 **break**;  
 **case** JointType.*GLOBAL\_HINGE*:  
 thisBoneOuterToInnerUV = thisBoneOuterToInnerUV.projectOntoPlane(thisBoneJoint.getHingeRotationAxis());  
 **break**;  
 **case** JointType.*LOCAL\_HINGE*:  
 **const** m = Mat3f.*createRotationMatrix*(**this**.**\_chain**[loop - 1].getDirectionUV());  
  
 **const** relativeHingeRotationAxis = m.times(thisBoneJoint.getHingeRotationAxis()).normalise();  
  
 thisBoneOuterToInnerUV = thisBoneOuterToInnerUV.projectOntoPlane(relativeHingeRotationAxis);  
 **break**;  
 }  
  
 **const** newStartLocation = target.*plus*(thisBoneOuterToInnerUV.times(thisBoneLength));  
  
 thisBone.setStartLocation(newStartLocation);  
  
 **if** (loop > 0) {  
 **this**.**\_chain**[loop - 1].setEndLocation(newStartLocation);  
 }  
 }  
  
 }  
  
 *// ---------- Обратный проход -----------* **for** (**let** loop = 0; loop < **this**.**\_chain**.**length**; ++loop) {  
 **const** thisBone: FabrikBone3D = **this**.**\_chain**[loop];  
 **const** thisBoneLength = thisBone.*getLength*();  
  
 **if** (loop != 0) {  
 **let** thisBoneInnerToOuterUV: Vec3f = thisBone.getDirectionUV();  
 **let** prevBoneInnerToOuterUV: Vec3f = **this**.**\_chain**[loop - 1].getDirectionUV();  
  
 **const** thisBoneJoint: FabrikJoint3D = thisBone.*getJoint*();  
 **const** jointType: JointType = thisBoneJoint.getJointType();  
 **if** (jointType == JointType.*BALL*) {  
 **const** angleBetweenDegs: **number** = Vec3f.*getAngleBetweenDegs*(prevBoneInnerToOuterUV, thisBoneInnerToOuterUV);  
 **const** constraintAngleDegs: **number** = thisBoneJoint.getBallJointConstraintDegs();  
  
 **if** (angleBetweenDegs > constraintAngleDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*getAngleLimitedUnitVectorDegs*(thisBoneInnerToOuterUV, prevBoneInnerToOuterUV, constraintAngleDegs);  
 }  
 }  
 **else if** (jointType == JointType.*GLOBAL\_HINGE*) {  
 **const** hingeRotationAxis: Vec3f = thisBoneJoint.getHingeRotationAxis();  
 thisBoneInnerToOuterUV = thisBoneInnerToOuterUV.projectOntoPlane(hingeRotationAxis);  
  
 **const** cwConstraintDegs = -thisBoneJoint.getHingeClockwiseConstraintDegs();  
 **const** acwConstraintDegs = thisBoneJoint.getHingeAnticlockwiseConstraintDegs();  
 **if** (!(Utils.*approximatelyEquals*(cwConstraintDegs, -FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.001)) &&  
 !(Utils.*approximatelyEquals*(acwConstraintDegs, FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.001))) {  
 **const** hingeReferenceAxis: Vec3f = thisBoneJoint.getHingeReferenceAxis();  
  
 **const** signedAngleDegs: **number** = Vec3f.*getSignedAngleBetweenDegs*(hingeReferenceAxis, thisBoneInnerToOuterUV, hingeRotationAxis);  
  
 **if** (signedAngleDegs > acwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, acwConstraintDegs, hingeRotationAxis).normalised();  
 } **else if** (signedAngleDegs < cwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, cwConstraintDegs, hingeRotationAxis).normalised();  
 }  
 }  
 } **else if** (jointType == JointType.*LOCAL\_HINGE*) {  
 **const** hingeRotationAxis: Vec3f = thisBoneJoint.getHingeRotationAxis();  
  
 **const** m: Mat3f = Mat3f.*createRotationMatrix*(prevBoneInnerToOuterUV);  
  
 **const** relativeHingeRotationAxis: Vec3f = m.times(hingeRotationAxis).normalise();  
  
 thisBoneInnerToOuterUV = thisBoneInnerToOuterUV.projectOntoPlane(relativeHingeRotationAxis);  
  
 **const** cwConstraintDegs = -thisBoneJoint.getHingeClockwiseConstraintDegs();  
 **const** acwConstraintDegs = thisBoneJoint.getHingeAnticlockwiseConstraintDegs();  
 **if** (!(Utils.*approximatelyEquals*(cwConstraintDegs, -FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.001)) &&  
 !(Utils.*approximatelyEquals*(acwConstraintDegs, FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.001))) {  
 **const** relativeHingeReferenceAxis: Vec3f = m.times(thisBoneJoint.getHingeReferenceAxis()).normalise();  
  
 **const** signedAngleDegs = Vec3f.*getSignedAngleBetweenDegs*(relativeHingeReferenceAxis, thisBoneInnerToOuterUV, relativeHingeRotationAxis);  
  
 **if** (signedAngleDegs > acwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(relativeHingeReferenceAxis, acwConstraintDegs, relativeHingeRotationAxis).normalise();  
 } **else if** (signedAngleDegs < cwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(relativeHingeReferenceAxis, cwConstraintDegs, relativeHingeRotationAxis).normalise();  
 }  
 }  
  
 }  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(thisBoneLength));  
  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (loop < **this**.**\_chain**.**length** - 1) {  
 **this**.**\_chain**[loop + 1].setStartLocation(newEndLocation);  
 }  
 } **else** {  
 **if** (**this**.**\_fixedBaseMode**) {  
 thisBone.setStartLocation(**this**.**\_fixedBaseLocation**);  
 } **else** {  
 thisBone.setStartLocation(thisBone.*getEndLocation*().*minus*(thisBone.getDirectionUV().times(thisBoneLength)));  
 }  
  
 **if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*NONE*) {  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBone.getDirectionUV().times(thisBoneLength));  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 } **else** {  
 **if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*GLOBAL\_ROTOR*) {  
 **let** thisBoneInnerToOuterUV: Vec3f = thisBone.getDirectionUV();  
  
 **const** angleBetweenDegs = Vec3f.*getAngleBetweenDegs*(**this**.**\_baseboneConstraintUV**, thisBoneInnerToOuterUV);  
 **const** constraintAngleDegs = thisBone.getBallJointConstraintDegs();  
  
 **if** (angleBetweenDegs > constraintAngleDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*getAngleLimitedUnitVectorDegs*(thisBoneInnerToOuterUV, **this**.**\_baseboneConstraintUV**, constraintAngleDegs);  
 }  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(thisBoneLength));  
  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 } **else if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*LOCAL\_ROTOR*) {  
 **let** thisBoneInnerToOuterUV = thisBone.getDirectionUV();  
  
 **const** angleBetweenDegs = Vec3f.*getAngleBetweenDegs*(**this**.**\_baseboneRelativeConstraintUV**, thisBoneInnerToOuterUV);  
 **const** constraintAngleDegs = thisBone.getBallJointConstraintDegs();  
 **if** (angleBetweenDegs > constraintAngleDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*getAngleLimitedUnitVectorDegs*(thisBoneInnerToOuterUV, **this**.**\_baseboneRelativeConstraintUV**, constraintAngleDegs);  
 }  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(thisBoneLength));  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 } **else if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*GLOBAL\_HINGE*) {  
 **const** thisJoint = thisBone.*getJoint*();  
 **const** hingeRotationAxis = thisJoint.getHingeRotationAxis();  
 **const** cwConstraintDegs = -thisJoint.getHingeClockwiseConstraintDegs();  
 **const** acwConstraintDegs = thisJoint.getHingeAnticlockwiseConstraintDegs();  
  
 **let** thisBoneInnerToOuterUV = thisBone.getDirectionUV().projectOntoPlane(hingeRotationAxis);  
  
 **if** (!(Utils.*approximatelyEquals*(cwConstraintDegs, -FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.01) &&  
 Utils.*approximatelyEquals*(acwConstraintDegs, FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.01))) {  
 **const** hingeReferenceAxis = thisJoint.getHingeReferenceAxis();  
 **const** signedAngleDegs = Vec3f.*getSignedAngleBetweenDegs*(hingeReferenceAxis, thisBoneInnerToOuterUV, hingeRotationAxis);  
  
 **if** (signedAngleDegs > acwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, acwConstraintDegs, hingeRotationAxis).normalise();  
 }  
 **else if** (signedAngleDegs < cwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, cwConstraintDegs, hingeRotationAxis).normalise();  
 }  
 }  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(thisBoneLength));  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 } **else if** (**this**.**\_baseboneConstraintType** == BaseboneConstraintType3D.*LOCAL\_HINGE*) {  
 **const** thisJoint = thisBone.*getJoint*();  
 **const** hingeRotationAxis = **this**.**\_baseboneRelativeConstraintUV**;  
 **const** cwConstraintDegs = -thisJoint.getHingeClockwiseConstraintDegs();  
 **const** acwConstraintDegs = thisJoint.getHingeAnticlockwiseConstraintDegs();  
  
 **let** thisBoneInnerToOuterUV = thisBone.getDirectionUV().projectOntoPlane(hingeRotationAxis);  
  
 **if** (!(Utils.*approximatelyEquals*(cwConstraintDegs, -FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.01) &&  
 Utils.*approximatelyEquals*(acwConstraintDegs, FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*, 0.01))) {  
 **const** hingeReferenceAxis = **this**.**\_baseboneRelativeReferenceConstraintUV**;  
 **const** signedAngleDegs = Vec3f.*getSignedAngleBetweenDegs*(hingeReferenceAxis, thisBoneInnerToOuterUV, hingeRotationAxis);  
  
 **if** (signedAngleDegs > acwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, acwConstraintDegs, hingeRotationAxis).normalise();  
 } **else if** (signedAngleDegs < cwConstraintDegs) {  
 thisBoneInnerToOuterUV = Vec3f.*rotateAboutAxisDegs*(hingeReferenceAxis, cwConstraintDegs, hingeRotationAxis).normalise();  
 }  
 }  
  
 **const** newEndLocation = thisBone.*getStartLocation*().*plus*(thisBoneInnerToOuterUV.times(thisBoneLength));  
 thisBone.setEndLocation(newEndLocation);  
  
 **if** (**this**.**\_chain**.**length** > 1) {  
 **this**.**\_chain**[1].setStartLocation(newEndLocation);  
 }  
 }  
 }  
 }  
 }  
  
 **this**.**\_lastTargetLocation**.set(target);  
  
 **return** Vec3f.*distanceBetween*(**this**.**\_chain**[**this**.**\_chain**.**length** - 1].*getEndLocation*(), target);  
 }  
  
 updateChainLength(): **void** {  
 **this**.**\_chainLength** = 0.0;  
  
 **for** (**let** aBone **of this**.**\_chain**) {  
 **this**.**\_chainLength** += aBone.*getLength*();  
 }  
 }  
  
 updateEmbeddedTarget(newEmbeddedTarget: Vec3f): **void** {  
 **if** (**this**.**\_useEmbeddedTarget**) {  
 **this**.**\_embeddedTarget**.set(newEmbeddedTarget);  
 } **else** {  
 **throw new** Error(**'This chain does not have embedded targets enabled - enable with setEmbeddedTargetMode(true).'**);  
 }  
 }  
  
 **private** cloneIkChain(): FabrikBone3D[] {  
 **const** clonedChain = [];  
  
 **for** (**let** aBone **of this**.**\_chain**) {  
 **const** bone = **new** FabrikBone3D();  
 bone.set(aBone);  
 clonedChain.push(bone);  
 }  
  
 **return** clonedChain;  
 }  
  
 getMaxIterationAttempts(): **number** {  
 **return this**.**\_maxIterationAttempts**;  
 }  
  
 getMinIterationChange(): **number** {  
 **return this**.**\_minIterationChange**;  
 }  
  
 getSolveDistanceThreshold(): **number** {  
 **return this**.**\_solveDistanceThreshold**;  
 }  
}

### FabrikJoint2D

**import** {ConstraintCoordinateSystem} **from './ConstraintCoordinateSystem'**;  
  
*/\*\*  
 \* Сустав, определяющий углы-ограничения между костями цепи.  
 \* <p>  
 \* FabrikJoint2D состоит из пары углов:  
 \* <ul><li>Угол-ограничение по часовой стрелке;</li>  
 \* <li>Угол-ограничение против часовой стрелки.</li></ul>  
 \* <p>  
 \* Они оба измеряются в градусах [0..180].  
 \* Значение по умолчанию для обоих углов - 180, что означает, что сустав не имеет ограничений.  
 \* Для установки собственных значений углов-ограничений можно воспользоваться конструктором, задать свойствам значения  
 \* непосредственно по ссылкам {****@link*** *#\_clockwiseConstraintDegs} и {****@link*** *#\_anticlockwiseConstraintDegs} или  
 \* воспользоваться методами-сеттерами {****@link*** *#setClockwiseConstraintDegs} and {****@link*** *#setAnticlockwiseConstraintDegs}.  
 \* Первый и третий метод наиболее предпочтительные, так как они осуществляют проверку задаваемых значений.  
 \* <p>  
 \* Каждая FabrikBone2D содержит в себе один FabrikJoint2D, который располагается в начале кости {****@code*** *mStartLocation}.  
 \* <p>  
 \* Сустав отделен от кости в самостоятельный класс в связи с тем, что существует несколько типов ограничений, что  
 \* делает функциональность сустава громоздкой и сложной для понимания внутри класса кости.  
 \*/***export class** FabrikJoint2D {  
 */\*\* Минимальный угол-ограничение для любого из направлений в градусах. Полностью обездвиживает кость. \*/* **static** *MIN\_2D\_CONSTRAINT\_ANGLE\_DEGS* = 0;  
  
 */\*\* Максимальный угол-ограничение для любого из направлений в градусах. Даёт кости полную свободу. \*/* **static** *MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS* = 180;  
  
 */\*\*  
 \* \_clockwiseConstraintDegs Угол в градусах, на который данный FabrikJoint2D может быть повернут по часовой стрелке  
 \* относительно предыдущей кости или мировой системы координат, в зависимости от выбранной системы координат.  
 \* <p>  
 \* Допустимые значения [0..180].  
 \*  
 \** ***@default*** *180.0  
 \*/* **private \_clockwiseConstraintDegs** = FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
  
 */\*\*  
 \* mAntiClockwiseContraintDegs Угол в градусах, на который данный FabrikJoint2D может быть повернут против часовой стрелки  
 \* относительно предыдущей кости или мировой системы координат, в зависимости от выбранной системы координат.  
 \* <p>  
 \* Допустимые значения [0..180].  
 \*  
 \** ***@default*** *180.0  
 \*/* **private \_anticlockwiseConstraintDegs** = FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
  
 */\*\*  
 \* \_constraintCoordinateSystem Система координат, относительно которой задаются углы-ограничения.  
 \* <p>  
 \* Может принимать значения из перечисления {****@link*** *#ConstraintCoordinateSystem}  
 \* <p>  
 \* Значение по умолчанию - ConstraintCoordinateSystem.LOCAL.  
 \*/* **private \_constraintCoordinateSystem** = ConstraintCoordinateSystem.*LOCAL*;  
  
 *// ---------- Constructor ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает три параметра угол-ограничение по часовой стрелке, против часовой стрелки и систему координат.  
 \* Все параметры опциональны, в случае отсутствия принимают значения по умолчанию.  
 \*  
 \** ***@param*** *clockwiseConstraintDegs Угол-ограничение по часовой стрелке в градусах.  
 \** ***@param*** *antiClockwiseConstraintDegs Угол-ограничение против часовой стрелки в градусах.  
 \** ***@param*** *constraintCoordSystem Система координат.  
 \*/* **constructor**(clockwiseConstraintDegs?: **number**, antiClockwiseConstraintDegs?: **number**, constraintCoordSystem?: ConstraintCoordinateSystem) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **return**;  
 **case** 3:  
 **this**.setClockwiseConstraintDegs(clockwiseConstraintDegs);  
 **this**.setAnticlockwiseConstraintDegs(antiClockwiseConstraintDegs);  
 **this**.**\_constraintCoordinateSystem** = constraintCoordSystem;  
 **return**;  
 **default**:  
 **throw** Error(**'Invalid FabrikJoint2D constructor params!'**);  
 }  
 }  
  
 *// ---------- Methods ----------  
  
 /\*\*  
 \* Копирует sourceJoint в данный сустав.  
 \*  
 \** ***@param*** *sourceJoint Сустав, значения которого будут скопированы.  
 \*/* set(sourceJoint: FabrikJoint2D): **void** {  
 **this**.setClockwiseConstraintDegs(sourceJoint.**\_clockwiseConstraintDegs**);  
 **this**.setAnticlockwiseConstraintDegs(sourceJoint.**\_anticlockwiseConstraintDegs**);  
 **this**.**\_constraintCoordinateSystem** = sourceJoint.**\_constraintCoordinateSystem**;  
 };  
  
 */\*\*  
 \* Задаёт угол-ограничение по часовой стрелке в градусах.  
 \* <p>  
 \* Угол-ограничение может принимать значения от 0 (не предусматривает движения), до 180 (полностью подвижен).  
 \** ***@param*** *angleDegs Угол-ограничение по часовой стрелке в градусах.  
 \*/* setClockwiseConstraintDegs (angleDegs: **number**): **void** {  
 **if** (angleDegs < FabrikJoint2D.*MIN\_2D\_CONSTRAINT\_ANGLE\_DEGS*) {  
 **this**.**\_clockwiseConstraintDegs** = FabrikJoint2D.*MIN\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
 } **else if** (angleDegs > FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*) {  
 **this**.**\_clockwiseConstraintDegs** = FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
 } **else** {  
 **this**.**\_clockwiseConstraintDegs** = angleDegs;  
 }  
 };  
  
 */\*\*  
 \* Возвращает угол-ограничение по часовой стрелке в градусах.  
 \*  
 \** ***@return*** *Угол-ограничение по часовой стрелке в градусах.  
 \*/  
 getClockwiseConstraintDegs* = (): **number** => **this**.**\_clockwiseConstraintDegs**;  
  
 */\*\*  
 \* Задаёт угол-ограничение против часовой стрелки в градусах.  
 \* <p>  
 \* Угол-ограничение может принимать значения от 0 (не предусматривает движения), до 180 (полностью подвижен).  
 \** ***@param*** *angleDegs Угол-ограничение против часовой стрелки в градусах.  
 \*/* setAnticlockwiseConstraintDegs (angleDegs: **number**): **void** {  
 **if** (angleDegs < FabrikJoint2D.*MIN\_2D\_CONSTRAINT\_ANGLE\_DEGS*) {  
 **this**.**\_anticlockwiseConstraintDegs** = FabrikJoint2D.*MIN\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
 }  
 **else if** (angleDegs > FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*) {  
 **this**.**\_anticlockwiseConstraintDegs** = FabrikJoint2D.*MAX\_2D\_CONSTRAINT\_ANGLE\_DEGS*;  
 } **else** {  
 **this**.**\_anticlockwiseConstraintDegs** = angleDegs;  
 }  
 };  
  
 */\*\*  
 \* Возвращает угол-ограничение против часовой стрелки в градусах.  
 \*  
 \** ***@return*** *Угол-ограничение против часовой стрелки в градусах.  
 \*/  
 getAnticlockwiseConstraintDegs* = (): **number** => **this**.**\_anticlockwiseConstraintDegs**;  
  
 */\*\*  
 \* Задаёт систему координат данного ограничения.  
 \*  
 \** ***@param*** *coordSystem Система координат.  
 \*/  
 setConstraintCoordinateSystem* = (coordSystem: ConstraintCoordinateSystem): **void** => {  
 **this**.**\_constraintCoordinateSystem** = coordSystem;  
 };  
  
 */\*\*  
 \* Возвращает систему координат данного ограничения.  
 \*  
 \** ***@return*** *Система координат.  
 \*/  
 getConstraintCoordinateSystem* = (): ConstraintCoordinateSystem => {  
 **return this**.**\_constraintCoordinateSystem**;  
 };  
}

### FabrikJoint3D

**import** {Utils} **from '../utils/Utils'**;  
**import** {JointType} **from './JointType'**;  
**import** {Vec3f} **from '../utils/Vec3f'**;  
  
*/\*\*  
 \* Сустав, определяющий углы-ограничения между костями цепи.  
 \* <p>  
 \* Существует три типа суставов:  
 \* <ul>  
 \* <li>JointType.BALL - сустав будет вращаться на один и тот же угол во все стороны относительно предыдущей кости  
 \* в цепи или относительно мировой системы координат, в случае если это начальная кость цепи,</li>  
 \* <li>JointType.GLOBAL\_HINGE - сустав, для которого заданы углы-ограничители относительно мировой системы координат  
 \* для выбранной оси по и против часовой стрелки,</li>  
 \* <li>JointType.LOCAL\_HINGE - сустав, для которого заданы углы-ограничители относительно предыдущей кости  
 \* для выбранной оси по и против часовой стрелки.</li>  
 \* </ul>  
 \* Шаровой сустав может вращаться в любую сторону относительно положения предыдущей кости в цепи.  
 \* Угол-ограничитель в 180 градусов будет давать суставу полную свободу движения, а угол в 0  
 \* градусов не предусматривает движения совсем, делая текущую кость продолжением предыдущей.  
 \* <p>  
 \* Шарнирный сустав может вращаться по и против часовой стрелки вдоль каждой оси на допустимое число градусов [0..180].  
 \* Локальный шарнир ведет отсчет градусов относительно предыдущей кости в цепи,  
 \* а глобальный - относительно мировой системы координат.  
 \* <p>  
 \* Каждая FabrikBone3D имеет один FabrikJoint3D, который располагается в начальной точке кости {****@code*** *mStartLocation}.  
 \*  
 \*/***export class** FabrikJoint3D {  
 */\*\* Минимальный угол-ограничение для любого из направлений в градусах. Полностью обездвиживает кость. \*/* **static** *MIN\_CONSTRAINT\_ANGLE\_DEGS*: **number** = 0.0;  
  
 */\*\* Максимальный угол-ограничение для любого из направлений в градусах. Даёт кости полную свободу. \*/* **static** *MAX\_CONSTRAINT\_ANGLE\_DEGS*: **number** = 180.0;  
  
 */\*\*  
 \* Угол-ограничение в градусах, на который позволяет повернуться сустав (для шаровых суставов).  
 \* <p>  
 \* Допустимые значения [0..180].  
 \*  
 \** ***@default*** *180.0  
 \*/* **private \_rotorConstraintDegs**: **number** = FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*;  
  
 */\*\*  
 \* Угол в градусах, на который сустав позволяет повернуться по часовой стрелке  
 \* относительно выбранной оси и системы координат (для шарнирных суставов).  
 \* <p>  
 \* Допустимые значения [0..180].  
 \*  
 \** ***@default*** *180.0  
 \*/* **private \_hingeClockwiseConstraintDegs**: **number** = FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*;  
  
 */\*\*  
 \* Угол в градусах, на который сустав позволяет повернуться против часовой стрелки  
 \* относительно выбранной оси и системы координат (для шарнирных суставов).  
 \* <p>  
 \* Допустимые значения [0..180].  
 \*  
 \** ***@default*** *180.0  
 \*/* **private \_hingeAnticlockwiseConstraintDegs**: **number** = FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*;  
  
 */\*\* Ось вращения шарнирного сустава. \*/* **private \_rotationAxisUV**: Vec3f = **new** Vec3f();  
  
 */\*\* Опорная ось шарнирного сустава. \*/* **private \_referenceAxisUV**: Vec3f = **new** Vec3f();  
  
 */\*\*  
 \* Тип сустава.  
 \* <p>  
 \* Допустимые значения - JointType.BALL, JointType.GLOBAL\_HINGE или JointType.LOCAL\_HINGE.  
 \*  
 \** ***@default*** *JointType.BALL  
 \*/* **private \_jointType**: JointType = JointType.*BALL*;  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает 2 возможных сочетания параметров:  
 \* <p>  
 \* Без параметров.  
 \* Все поля получают занчения по умолчанию.  
 \* Задать недефолтные значения можно через методы  
 \* setAsGlobalHinge(Vec3f, number, number, Vec3f) и  
 \* setAsLocalHinge(Vec3f, number, number, Vec3f).  
 \* <p>  
 \* Конструктор копирования.  
 \* Создаёт глубокую копию сустава.  
 \*  
 \** ***@param*** *source Сустав источник.  
 \*/* **constructor**(source?: FabrikJoint3D) { **if** (source) **this**.set(source); }  
  
 */\*\*  
 \* Делает из данного сустава глубокую копию сустава-источника.  
 \*  
 \** ***@param*** *source Сустав источник.  
 \*/* set(source: FabrikJoint3D): **void** {  
 **this**.**\_jointType** = source.**\_jointType**;  
 **this**.**\_rotorConstraintDegs** = source.**\_rotorConstraintDegs**;  
 **this**.**\_hingeClockwiseConstraintDegs** = source.**\_hingeClockwiseConstraintDegs**;  
 **this**.**\_hingeAnticlockwiseConstraintDegs** = source.**\_hingeAnticlockwiseConstraintDegs**;  
  
 **this**.**\_rotationAxisUV**.set(source.**\_rotationAxisUV**);  
 **this**.**\_referenceAxisUV**.set(source.**\_referenceAxisUV**);  
 }  
  
 */\*\*  
 \* Задаёт данному суставу шаровой тип.  
 \*  
 \** ***@param*** *constraintAngleDegs Максимальный допустимый угол в градусах между костью сустава и предыдущей костью в цепи.  
 \*/* setAsBallJoint(constraintAngleDegs: **number**): **void** {  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(constraintAngleDegs);  
  
 **this**.**\_rotorConstraintDegs** = constraintAngleDegs;  
 **this**.**\_jointType** = JointType.*BALL*;  
 }  
  
 */\*\*  
 \* Задаёт данному суставу шарнирный тип.  
 \*  
 \** ***@param*** *jointType Тип сустава, допустимы и BALL, и GLOBAL\_HINGE, и LOCAL\_HINGE.  
 \** ***@param*** *rotationAxis Ось вращения шарнира.  
 \** ***@param*** *clockwiseConstraintDegs Угол ограничение по часовой стрелке относительно опорной оси.  
 \** ***@param*** *anticlockwiseConstraintDegs Угол ограничение против часовой стрелки относительно опорной оси.  
 \** ***@param*** *referenceAxis Опорная ось.  
 \*/* setHingeJoint(jointType: JointType, rotationAxis: Vec3f, clockwiseConstraintDegs: **number**, anticlockwiseConstraintDegs: **number**, referenceAxis: Vec3f): **void** {  
 *// Убедимся, что ось вращения и опорная ось образуют плоскость (т. к. они могут быть перпендикулярны и их скалярное произведение равно 0)* **if** ( !Utils.*approximatelyEquals*( Vec3f.*dotProduct*(rotationAxis, referenceAxis), 0.0, 0.01) ) {  
 **const** angleDegs = Vec3f.*getAngleBetweenDegs*(rotationAxis, referenceAxis);  
 **throw new** Error(**'The reference axis must be in the plane of the hinge rotation axis - angle between them is currently: '** + angleDegs);  
 }  
  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(clockwiseConstraintDegs);  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(anticlockwiseConstraintDegs);  
 FabrikJoint3D.*\_validateAxis*(rotationAxis);  
 FabrikJoint3D.*\_validateAxis*(referenceAxis);  
  
 **this**.**\_hingeClockwiseConstraintDegs** = clockwiseConstraintDegs;  
 **this**.**\_hingeAnticlockwiseConstraintDegs** = anticlockwiseConstraintDegs;  
 **this**.**\_jointType** = jointType;  
 **this**.**\_rotationAxisUV**.set( rotationAxis.normalised() );  
 **this**.**\_referenceAxisUV**.set( referenceAxis.normalised() );  
 }  
  
 */\*\*  
 \* Задаёт угол ограничение по часовой стрелке для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибки, если угол выходит за границы допустимых значений и  
 \* если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *angleDegs Угол ограничение по часовой стрелке в градусах.  
 \*/* setHingeJointClockwiseConstraintDegs(angleDegs: **number**): **void** {  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(angleDegs);  
  
 **if** (**this**.**\_jointType** !== JointType.*BALL*) {  
 **this**.**\_hingeClockwiseConstraintDegs** = angleDegs;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have hinge constraint angles.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает угол-ограничение по часовой стрелке для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шарнирный.  
 \*  
 \** ***@return*** *Угол-ограничение по часовой стрелке в градусах.  
 \*/* getHingeClockwiseConstraintDegs(): **number** {  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **return this**.**\_hingeClockwiseConstraintDegs**;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have hinge constraint angles.'**);  
 }  
 }  
  
 */\*\*  
 \* Задаёт угол ограничение против часовой стрелки для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибки, если угол выходит за границы допустимых значений и  
 \* если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *angleDegs Угол ограничение против часовой стрелки в градусах.  
 \*/* setHingeJointAnticlockwiseConstraintDegs(angleDegs: **number**): **void** {  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(angleDegs);  
  
 **if** (**this**.**\_jointType** !== JointType.*BALL*) {  
 **this**.**\_hingeAnticlockwiseConstraintDegs** = angleDegs;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have hinge constraint angles.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает угол-ограничение против часовой стрелки для шарнирного сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шарнирный.  
 \*  
 \** ***@return*** *Угол-ограничение против часовой стрелки в градусах.  
 \*/* getHingeAnticlockwiseConstraintDegs() {  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **return this**.**\_hingeAnticlockwiseConstraintDegs**;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have hinge constraint angles.'**);  
 }  
 }  
  
 */\*\*  
 \* Задаёт угол-ограничение для шарового сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шаровой или  
 \* угол выходит за границы допустимых значений.  
 \*  
 \** ***@param*** *angleDegs Угол-ограничение в градусах.  
 \*/* setBallJointConstraintDegs(angleDegs: **number**): **void** {  
 FabrikJoint3D.*\_validateConstraintAngleDegs*(angleDegs);  
  
 **if** (**this**.**\_jointType** === JointType.*BALL*) {  
 **this**.**\_rotorConstraintDegs** = angleDegs;  
 } **else** {  
 **throw new** Error(**'This joint is of type: '** + **this**.**\_jointType** + **' - only joints of type JointType.BALL have a ball joint constraint angle.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает угол ограничение шарового сустава.  
 \* <p>  
 \* Вызывает ошибку, если тип сустава не шаровой  
 \*  
 \** ***@return*** *Угол-ограничение в градусах.  
 \*/* getBallJointConstraintDegs(): **number** {  
 **if** (**this**.**\_jointType** === JointType.*BALL*) {  
 **return this**.**\_rotorConstraintDegs**;  
 }  
 **else** {  
 **throw new** Error(**'This joint is not of type JointType.BALL - it does not have a ball joint constraint angle.'**);  
 }  
 }  
  
 */\*\*  
 \* Задаёт ось вращения шарнирного сустава как нормализованную версию исходной оси.  
 \* <p>  
 \* Вызывает ошибки, если переданная ось имеет нулевую длину и если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *axis Ось вращения.  
 \*/* setHingeRotationAxis(axis: Vec3f): **void** {  
 FabrikJoint3D.*\_validateAxis*(axis);  
  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **this**.**\_rotationAxisUV**.set( axis.normalised() );  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have a hinge rotation axis.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает опорную ось шарнирного сустава.  
 \* <p>  
 \* Если тип сустава не шарнирный, выбрасывается ошибка.  
 \*  
 \** ***@return*** *Опорная ось.  
 \*/* getHingeReferenceAxis(): Vec3f {  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **return this**.**\_referenceAxisUV**;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have a hinge reference axis.'**);  
 }  
 }  
  
 */\*\*  
 \* Задаёт опорную ось шарнирного сустава, относительно которой измеряется угол вращения сустава.  
 \* <p>  
 \* Вызывает ошибки, если переданная ось имеет нулевую длину и если тип сустава не шарнирный.  
 \*  
 \** ***@param*** *referenceAxis Опорная ось.  
 \*/* setHingeReferenceAxis(referenceAxis: Vec3f): **void** {  
 FabrikJoint3D.*\_validateAxis*(referenceAxis);  
  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **this**.**\_referenceAxisUV**.set( referenceAxis.normalised() );  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have a hinge reference axis.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает ось вращения шарнирного сустава.  
 \* <p>  
 \* Если у сустава тип JointType.BALL (т.е. он не шарнирный) бросам ошибку.  
 \*  
 \** ***@return*** *Ось вращения шарнирного сустава.  
 \*/* getHingeRotationAxis(): Vec3f {  
 **if** ( **this**.**\_jointType** !== JointType.*BALL* ) {  
 **return this**.**\_rotationAxisUV**;  
 } **else** {  
 **throw new** Error(**'Joint type is JointType.BALL - it does not have a hinge rotation axis.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает тип сустава.  
 \* <p>  
 \* Допустимые значения: JointType.BALL, JointType.GLOBAL\_HINGE, JointType.LOCAL\_HINGE.  
 \*  
 \** ***@return*** *Тип сустава.  
 \*/* getJointType(): JointType { **return this**.**\_jointType**; }  
  
 *// ---------- Private Methods ----------* **static** *\_validateConstraintAngleDegs*(angleDegs: **number**): **void** {  
 **if** (angleDegs < FabrikJoint3D.*MIN\_CONSTRAINT\_ANGLE\_DEGS* || angleDegs > FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS*) {  
 **throw new** Error(**'Constraint angles must be within the range '** + FabrikJoint3D.*MIN\_CONSTRAINT\_ANGLE\_DEGS* + **' to '** + FabrikJoint3D.*MAX\_CONSTRAINT\_ANGLE\_DEGS* + **' inclusive.'**);  
 }  
 }  
  
 **static** *\_validateAxis*(axis: Vec3f): **void** {  
 **if** ( axis.*length*() <= 0.0 ) {  
 **throw new** Error(**'Provided axis is illegal - it has a magnitude of zero.'**);  
 }  
 }  
}

### FabrikStructure2D

**import** {Vec2f} **from '../utils/Vec2f'**;  
**import** {FabrikChain2D} **from './FabrikChain2D'**;  
**import** {BoneConnectionPoint} **from './BoneConnectionPoint'**;  
**import** {BaseboneConstraintType2D} **from './BaseboneConstraintType2D'**;  
  
*/\*\*  
 \* Набор 2D цепей.  
 \*\*/***export class** FabrikStructure2D {  
 **private static** *UP*: Vec2f = **new** Vec2f(0.0, 1.0);  
  
 *// ---------- Private Properties ----------  
  
 /\*\* Название структуры. \*/* **private \_name**: **string** = **''**;  
  
 */\*\* Список цепей. \*/* **private \_chains**: FabrikChain2D[] = [];  
  
 *// --------- Public Methods ----------  
  
 /\*\* Конструктор.  
 \* Принимает два варианта значений:  
 \* Конструктор без параметров - задает всем полям значения по умолчанию.  
 \*  
 \* Один параметр. Задает имя структуры.  
 \*  
 \** ***@param*** *name Имя структуры.  
 \* \*/* **constructor**(name: **string** = **''**) {  
 **this**.**\_name** = name;  
 }  
  
 */\*\*  
 \* Задаёт имя структуре.  
 \*  
 \** ***@param*** *name Желаемое имя.  
 \*/* **public** setName(name: **string**): **void** { **this**.**\_name** = name; }  
  
 */\*\*  
 \* Решает задачу ИК для всех цепей структуры.  
 \* <p>  
 \* Все цепи перемещаются к единой целевой позиции за исключением тех, для которых задан embeddedTargetMode.  
 \*  
 \** ***@param*** *newTargetLocation Целевая позиция для всех эффекторов структуры.  
 \*/* **public** solveForTarget(newTargetLocation: Vec2f): **void** {  
 **const** nu\_chains = **this**.**\_chains**.**length**;  
 **let** hostChainNumber: **number**;  
 **let** thisChain: FabrikChain2D;  
  
  
 **for** (**let** loop = 0; loop < nu\_chains; ++loop) {  
 thisChain = **this**.**\_chains**[loop];  
  
 hostChainNumber = thisChain.getConnectedChainNumber();  
  
 **const** constraintType = thisChain.getBaseboneConstraintType();  
  
 **if** (hostChainNumber !== -1 && constraintType !== BaseboneConstraintType2D.*GLOBAL\_ABSOLUTE*) {  
 **const** hostBone = **this**.**\_chains**[hostChainNumber].*getBone*(**this**.**\_chains**[loop].*getConnectedBoneNumber*());  
  
 **if** (thisChain.getBoneConnectionPoint() == BoneConnectionPoint.*START*) {  
 thisChain.setBaseLocation(hostBone.*getStartLocation*());  
 }  
 **else** {  
 thisChain.setBaseLocation(hostBone.*getEndLocation*());  
 }  
  
 **const** hostBoneUV = hostBone.getDirectionUV();  
 **if** (constraintType == BaseboneConstraintType2D.*LOCAL\_RELATIVE*) {  
 **this**.**\_chains**[loop].setBaseboneConstraintUV(hostBoneUV);  
 }  
 **else if** (constraintType == BaseboneConstraintType2D.*LOCAL\_ABSOLUTE*) {  
 **const** angleDegs = FabrikStructure2D.*UP*.getSignedAngleDegsTo(hostBoneUV);  
  
 **const** relativeConstraintUV = Vec2f.*rotateDegs*(thisChain.getBaseboneConstraintUV(), angleDegs);  
  
 thisChain.setBaseboneRelativeConstraintUV(relativeConstraintUV);  
 }  
 }  
  
 **if** (!thisChain.getEmbeddedTargetMode()) {  
 thisChain.solveForTarget(newTargetLocation);  
 }  
 **else** {  
 thisChain.solveForEmbeddedTarget();  
 }  
  
 }  
  
 }  
  
 */\*\*  
 \* Добавляет цепь в структуру.  
 \*  
 \** ***@param*** *chain Новая цепь.  
 \*/* addChain(chain: FabrikChain2D): **void** {  
 **this**.**\_chains**.push(chain);  
 }  
 */\*\*  
 \* Добавляет цепь в структуру, присоединяя ее к существующей в структуре цепи.  
 \*  
 \** ***@param*** *newChain Новая цепь.  
 \** ***@param*** *existingChainNumber Номер цепи, к которой необходимо присоединить новую цепь.  
 \** ***@param*** *existingBoneNumber Номер кости, к которой необходимо присоединить новую цепь.  
 \** ***@param*** *boneConnectionPoint К началу или к концу кости присоединять.  
 \*/* connectChain(chain: FabrikChain2D, chainNumber: **number**, boneNumber: **number**, boneConnectionPoint: BoneConnectionPoint, shouldCalcCoordinates: **boolean**): **void** {  
 chain.setBoneConnectionPoint(boneConnectionPoint);  
 **if** (chainNumber >= **this**.**\_chains**.**length**) {  
 **throw new** Error(**'Cannot connect to chain '** + chainNumber + **' - no such chain (remember that chains are zero indexed).'**);  
 }  
  
 **if** (boneNumber >= **this**.**\_chains**[chainNumber].*getNumBones*()) {  
 **throw new** Error(**'Cannot connect to bone '** + boneNumber + **' of chain '** + chainNumber + **' - no such bone (remember that bones are zero indexed).'**);  
 }  
  
 **const** relativeChain = **new** FabrikChain2D(chain);  
 relativeChain.setConnectedChainNumber(chainNumber);  
 relativeChain.setConnectedBoneNumber(boneNumber);  
  
 **const** connectionPoint = chain.getBoneConnectionPoint();  
 **let** connectionLocation;  
 **if** (connectionPoint == BoneConnectionPoint.*START*) {  
 connectionLocation = **this**.**\_chains**[chainNumber].*getBone*(boneNumber).*getStartLocation*();  
 } **else** {  
 connectionLocation = **this**.**\_chains**[chainNumber].*getBone*(boneNumber).*getEndLocation*();  
 }  
 relativeChain.setBaseLocation(connectionLocation);  
  
 relativeChain.setFixedBaseMode(**true**);  
  
 **if** (shouldCalcCoordinates) {  
 **for** (**let** loop = 0; loop < chain.getNumBones(); ++loop) {  
 **const** origStart = relativeChain.getBone(loop).*getStartLocation*();  
 **const** origEnd = relativeChain.getBone(loop).*getEndLocation*();  
  
 **const** translatedStart = origStart.*plus*(connectionLocation);  
 **const** translatedEnd = origEnd.*plus*(connectionLocation);  
  
 relativeChain.getBone(loop).setStartLocation(translatedStart);  
 relativeChain.getBone(loop).setEndLocation(translatedEnd);  
 }  
 }  
  
 **this**.addChain(relativeChain);  
 }  
  
 */\*\*  
 \* Возвращает количество цепей в структуре.  
 \*/* getNumChains(): **number** { **return this**.**\_chains**.**length**; }  
  
 */\*\*  
 \* Возвращает цепь структуры по индексу.  
 \*  
 \** ***@param*** *chainNumber Индекс цепи.  
 \*/* getChain(chainNumber: **number**): FabrikChain2D { **return this**.**\_chains**[chainNumber]; }  
  
 */\*\*  
 \* Возвращает имя структуры.  
 \*  
 \** ***@return*** *\*/* getName(): **string** {  
 **return this**.**\_name**;  
 }  
}

### FabrikStructure3D

**import** {FabrikChain3D} **from './FabrikChain3D'**;  
**import** {FabrikBone3D} **from './FabrikBone3D'**;  
**import** {BoneConnectionPoint} **from './BoneConnectionPoint'**;  
**import** {Vec3f} **from '../utils/Vec3f'**;  
**import** {BaseboneConstraintType3D} **from './BaseboneConstraintType3D'**;  
**import** {Mat3f} **from '../utils/Mat3f'**;  
  
*/\*\*  
 \* Набор 3D цепей.  
 \*\*/***export class** FabrikStructure3D {  
 *// ---------- Private Properties ----------  
  
 /\*\* Название структуры. \*/* **private \_name**: **string** = **''**;  
  
 */\*\* Список цепей. \*/* **private \_chains**: FabrikChain3D[] = [];  
  
 *// --------- Public Methods ----------  
  
 /\*\*  
 \* Конструктор.  
 \* Принимает два варианта значений:  
 \*  
 \* Конструктор без параметров - задает всем полям значения по умолчанию.  
 \*  
 \* Один параметр. Задает имя структуры.  
 \** ***@param*** *name Имя структуры.  
 \* \*/* **constructor**(name: **string** = **''**) {  
 **this**.**\_name** = name;  
 }  
  
 */\*\*  
 \* Решает задачу ИК для всех цепей структуры.  
 \* <p>  
 \* Все цепи перемещаются к единой целевой позиции за исключением тех, для которых задан embeddedTargetMode.  
 \*  
 \** ***@param*** *newTargetLocation Целевая позиция для всех эффекторов структуры.  
 \*/* **public** solveForTarget(newTargetLocation: Vec3f): **void** {  
 **const** nu\_chains = **this**.**\_chains**.**length**;  
 **let** connectedChainNumber;  
  
 **for** (**let** loop = 0; loop < nu\_chains; ++loop) {  
 **const** thisChain = **this**.**\_chains**[loop];  
 connectedChainNumber = thisChain.*getConnectedChainNumber*();  
  
 **if** (connectedChainNumber === -1) {  
 **if** (!thisChain.getEmbeddedTargetMode()) {  
 thisChain.solveForTarget(newTargetLocation);  
 } **else** {  
 thisChain.solveForEmbeddedTarget();  
 }  
 } **else** {  
 **const** hostChain: FabrikChain3D = **this**.**\_chains**[connectedChainNumber];  
 **const** hostBone: FabrikBone3D = hostChain.*getBone*(thisChain.*getConnectedBoneNumber*());  
 **if** (hostBone.*getBoneConnectionPoint*() == BoneConnectionPoint.*START*) {  
 thisChain.setBaseLocation(hostBone.*getStartLocation*());  
 } **else** {  
 thisChain.setBaseLocation(hostBone.*getEndLocation*());  
 }  
  
 **const** constraintType: BaseboneConstraintType3D = thisChain.getBaseboneConstraintType();  
 **switch** (constraintType) {  
 **case** BaseboneConstraintType3D.*NONE*:  
 **case** BaseboneConstraintType3D.*GLOBAL\_ROTOR*:  
 **case** BaseboneConstraintType3D.*GLOBAL\_HINGE*:  
 **break**;  
  
 **case** BaseboneConstraintType3D.*LOCAL\_ROTOR*:  
 **case** BaseboneConstraintType3D.*LOCAL\_HINGE*: {  
 **const** connectionBoneMatrix: Mat3f = Mat3f.*createRotationMatrix*(hostBone.getDirectionUV());  
 **const** relativeBaseboneConstraintUV: Vec3f = connectionBoneMatrix.times(thisChain.getBaseboneConstraintUV()).normalised();  
  
 thisChain.\_setBaseboneRelativeConstraintUV(relativeBaseboneConstraintUV);  
  
 **if** (constraintType == BaseboneConstraintType3D.*LOCAL\_HINGE*) {  
 thisChain.\_setBaseboneRelativeReferenceConstraintUV(connectionBoneMatrix.times(thisChain.*getBone*(0).*getJoint*().getHingeReferenceAxis()));  
 }  
 **break**;  
 }  
  
 }  
 **if** (!thisChain.getEmbeddedTargetMode()) {  
 thisChain.solveForTarget(newTargetLocation);  
 } **else** {  
 thisChain.solveForEmbeddedTarget();  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Добавляет цепь в структуру.  
 \*  
 \** ***@param*** *chain Новая цепь.  
 \*/* addChain(chain: FabrikChain3D): **void** {  
 **this**.**\_chains**.push(chain);  
 }  
  
 */\*\*  
 \* Удаляет цепь из структуры.  
 \*  
 \** ***@param*** *chainIndex Индекс удаляемой цепи.  
 \*/* removeChain(chainIndex): **void** {  
 **this**.**\_chains**.splice(chainIndex, 1);  
 }  
  
 */\*\*  
 \* Добавляет цепь в структуру, присоединяя ее к существующей в структуре цепи.  
 \*  
 \** ***@param*** *newChain Новая цепь.  
 \** ***@param*** *existingChainNumber Номер цепи, к которой необходимо присоединить новую цепь.  
 \** ***@param*** *existingBoneNumber Номер кости, к которой необходимо присоединить новую цепь.  
 \** ***@param*** *boneConnectionPoint К началу или к концу кости присоединять.  
 \*/* connectChain(newChain: FabrikChain3D, existingChainNumber: **number**, existingBoneNumber: **number**, boneConnectionPoint: BoneConnectionPoint, shouldCalcCoordinates: **boolean**): **void** {  
 **if** (existingChainNumber > **this**.**\_chains**.**length**) {  
 **throw new** Error(**'Cannot connect to chain '** + existingChainNumber + **' - no such chain (remember that chains are zero indexed).'**);  
 }  
  
 **if** (existingBoneNumber > **this**.**\_chains**[existingChainNumber].*getNumBones*()) {  
 **throw new** Error(**'Cannot connect to bone '** + existingBoneNumber + **' of chain '** + existingChainNumber + **' - no such bone (remember that bones are zero indexed).'**);  
 }  
  
 **const** relativeChain: FabrikChain3D = **new** FabrikChain3D(newChain);  
  
 relativeChain.connectToStructure(**this**, existingChainNumber, existingBoneNumber);  
  
 **this**.getChain(existingChainNumber).*getBone*(existingBoneNumber).setBoneConnectionPoint(boneConnectionPoint);  
 **let** connectionLocation: Vec3f;  
 **if** (boneConnectionPoint == BoneConnectionPoint.*START*) {  
 connectionLocation = **this**.**\_chains**[existingChainNumber].*getBone*(existingBoneNumber).*getStartLocation*();  
 } **else** {  
 connectionLocation = **this**.**\_chains**[existingChainNumber].*getBone*(existingBoneNumber).*getEndLocation*();  
 }  
 relativeChain.setBaseLocation(connectionLocation);  
  
 relativeChain.setFixedBaseMode(**true**);  
  
 **if** (shouldCalcCoordinates) {  
 **for** (**let** loop = 0; loop < relativeChain.*getNumBones*(); ++loop) {  
 **const** origStart: Vec3f = relativeChain.*getBone*(loop).*getStartLocation*();  
 **const** origEnd: Vec3f = relativeChain.*getBone*(loop).*getEndLocation*();  
  
 **const** translatedStart: Vec3f = origStart.*plus*(connectionLocation);  
 **const** translatedEnd: Vec3f = origEnd.*plus*(connectionLocation);  
  
 relativeChain.*getBone*(loop).setStartLocation(translatedStart);  
 relativeChain.*getBone*(loop).setEndLocation(translatedEnd);  
 }  
 }  
  
 **this**.addChain(relativeChain);  
 }  
  
 */\*\*  
 \* Возвращает количество цепей в структуре.  
 \*/* getNumChains(): **number** {  
 **return this**.**\_chains**.**length**;  
 }  
  
 */\*\*  
 \* Возвращает цепь структуры по индексу.  
 \*  
 \** ***@param*** *chainNumber Индекс цепи.  
 \*/* getChain(chainNumber: **number**): FabrikChain3D {  
 **return this**.**\_chains**[chainNumber];  
 }  
  
 */\*\*  
 \* Задаёт имя структуре.  
 \*  
 \** ***@param*** *name Желаемое имя.  
 \*/* setName(name: **string**): **void** {  
 **this**.**\_name** = name;  
 }  
  
 */\*\*  
 \* Возвращает имя структуры.  
 \*  
 \** ***@return*** *\*/* getName(): **string** {  
 **return this**.**\_name**;  
 }  
}

### JointType

*/\*\*  
 \* Тип сустава.  
 \*  
 \* <ul><li><strong>JointType.BALL</strong> - тип сустава, имеющий один угол-ограничение,  
 \* на который он может поворачиваться во всех направлениях. Если угол-ограничитель равен 180 -  
 \* сустав может быть повернут куда угодно, в противном случае, если угол-ограничитель равен 0 -  
 \* сустав не может двигаться вовсе.</li>  
 \* <li><strong>JointType.GLOBAL\_HINGE</strong> - тип сустава, при котором ограничения задаются  
 \* на вращение в определенном направлении, относительно мировой системы координат. По выбранной оси  
 \* задаётся 2 значения - по и против часовой стрелки - на которые сустав позволяет поворачиваться.  
 \* Все значения указываются в пределах от 0 до 180 градусов.</li>  
 \* <li><strong>JointType.LOCAL\_HINGE</strong> - тип сустава, при котором ограничения задаются  
 \* а вращение в определенном направлении, относительно предыдущей кости. По выбранной оси  
 \* задаётся 2 значения - по и против часовой стрелки - на которые сустав позволяет поворачиваться.  
 \* Все значения указываются в пределах от 0 до 180 градусов.</li>  
 \* </ul>  
 \*/***export enum** JointType {*BALL*, *GLOBAL\_HINGE*, *LOCAL\_HINGE*}

## Модуль визуализации

### Camera

*/\*\*  
 \* Камера в сцене.  
 \*/***export class** Camera {  
 **projectionMatrix**;  
 **transform**;  
 **viewMatrix**;  
  
 **constructor**(gl, fov = 45, near = 0.1, far = 100.0) {  
 **this**.**projectionMatrix** = **new** Float32Array(16);  
 **const** ratio = gl.**canvas**.**width** / gl.**canvas**.**height**;  
 Mat4f.*perspective*(**this**.**projectionMatrix**, fov, ratio, near, far);  
  
 **this**.**transform** = **new** Transform();  
 **this**.**viewMatrix** = **new** Float32Array(16);  
 }  
  
 panX(v) {  
 **this**.updateViewMatrix();  
 **this**.**transform**.**position**.**x** += v;  
 }  
  
 panY(v) {  
 **this**.updateViewMatrix();  
 **this**.**transform**.**position**.**y** += **this**.**transform**.**up**[1] \* v;  
 }  
  
 panZ(v) {  
 **this**.updateViewMatrix();  
 **this**.**transform**.**position**.**z** += v;  
 }  
  
 updateViewMatrix() {  
  
 **this**.**transform**.**matView**.reset()  
 .rotateX(**this**.**transform**.**rotation**.**x** \* Utils.*DEGS\_TO\_RADS*)  
 .rotateY(**this**.**transform**.**rotation**.**y** \* Utils.*DEGS\_TO\_RADS*)  
 .vtranslate(**this**.**transform**.**position**);  
  
  
  
 **this**.**transform**.updateDirection();  
  
 Mat4f.*invert*(**this**.**viewMatrix**, **this**.**transform**.**matView**.**raw**);  
 **return this**.**viewMatrix**;  
 }  
}

### CameraController

*/\*\*  
 \* Контроллер, меняющий положение камеры в ответ на действия пользователя.  
 \*/***export class** CameraController {  
 **canvas**;  
 **camera**;  
 **rotateRate**= -300;  
 **panRate** = 5;  
 **zoomRate** = 200;  
 **offsetX**;  
 **offsetY**;  
 **prevX** = 0;  
 **prevY** = 0;  
 **onUpHandler**;  
 **onMoveHandler**;  
  
 **constructor**(gl, camera) {  
 **const** oThis = **this**;  
 **const** box = gl.**canvas**.getBoundingClientRect();  
 **this**.**canvas** = gl.**canvas**;  
 **this**.**camera** = camera;  
  
 **this**.**offsetX** = box.**left**;  
 **this**.**offsetY** = box.**top**;  
  
 **if** (DIMENSION !== 2) {  
 **this**.**onUpHandler** = **function** () {  
 oThis.onMouseUp();  
 };  
 **this**.**onMoveHandler** = **function** (e) {  
 oThis.onMouseMove(e);  
 };  
  
 **this**.**canvas**.addEventListener(**'mousedown'**, **function** (e) {  
 oThis.onMouseDown(e);  
 });  
 }  
  
 **this**.**canvas**.addEventListener(**'mousewheel'**, **function** (e) {  
 oThis.onMouseWheel(e);  
 });  
  
 }  
   
 onMouseDown(e) {  
 **this**.**prevX** = e.**pageX** - **this**.**offsetX**;  
 **this**.**prevY** = e.**pageY** - **this**.**offsetY**;  
  
 **this**.**canvas**.addEventListener(**'mouseup'**, **this**.**onUpHandler**);  
 **this**.**canvas**.addEventListener(**'mousemove'**, **this**.**onMoveHandler**);  
 }  
  
 onMouseUp() {  
 **this**.**canvas**.removeEventListener(**'mouseup'**, **this**.**onUpHandler**);  
 **this**.**canvas**.removeEventListener(**'mousemove'**, **this**.**onMoveHandler**);  
 }  
  
 onMouseWheel(e) {  
 **let** delta = Math.max(-1, Math.min(1, (e.wheelDelta || -e.**detail**)));   
 **this**.**camera**.panZ(delta \* (**this**.**zoomRate** / **this**.**canvas**.**height**));   
 }  
  
 onMouseMove(e) {  
 **let** x = e.**pageX** - **this**.**offsetX**,   
 y = e.**pageY** - **this**.**offsetY**,  
 dx = x - **this**.**prevX**,   
 dy = y - **this**.**prevY**;  
  
   
 **if** (!e.shiftKey || DIMENSION === 2) {  
 **this**.**camera**.**transform**.**rotation**.**y** += dx \* (**this**.**rotateRate** / **this**.**canvas**.**width**);  
 **this**.**camera**.**transform**.**rotation**.**x** += dy \* (**this**.**rotateRate** / **this**.**canvas**.**height**);  
 } **else** {  
 **this**.**camera**.panX(-dx \* (**this**.**panRate** / **this**.**canvas**.**width**));  
 **this**.**camera**.panY(dy \* (**this**.**panRate** / **this**.**canvas**.**height**));  
 }  
  
 **this**.**prevX** = x;  
 **this**.**prevY** = y;  
 }  
}

### Modal

**import** {Transform} **from './Transform'**;  
  
*/\*\*  
 \* Универсальное представление отрисовываемого объекта.  
 \* Содержит в себе исходные данные об объекте, а также его положение относительно мировых координат.  
 \*/***export class** Modal {  
 **transform**;  
 **mesh**;  
  
 **constructor**(meshData) {  
 **this**.**transform** = **new** Transform();  
 **this**.**mesh** = meshData;  
 }  
   
 setScale(x, y, z) {  
 **this**.**transform**.**scale**.set(x, y, z);  
 **return this**;  
 }  
  
 setPosition(x, y, z) {  
 **this**.**transform**.**position**.set(x, y, z);  
 **return this**;  
 }  
  
 setRotation(x, y, z) {  
 **this**.**transform**.**rotation**.set(x, y, z);  
 **return this**;  
 }  
  
 addScale(x, y, z) {  
 **this**.**transform**.**scale**.**x** += x;  
 **this**.**transform**.**scale**.**y** += y;  
 **this**.**transform**.**scale**.**z** += z;  
 **return this**;  
 }  
  
 addPosition(x, y, z) {  
 **this**.**transform**.**position**.**x** += x;  
 **this**.**transform**.**position**.**y** += y;  
 **this**.**transform**.**position**.**z** += z;  
 **return this**;  
 }  
  
 addRotation(x, y, z) {  
 **this**.**transform**.**rotation**.**x** += x;  
 **this**.**transform**.**rotation**.**y** += y;  
 **this**.**transform**.**rotation**.**z** += z;  
 **return this**;  
 }  
  
 preRender() {  
 **this**.**transform**.updateMatrix();  
 **return this**;  
 }  
}

### Primatives.Primitive

Primatives.**Primitive** = **class** {  
 **static** *createModal*(gl, name, verts, colors) {  
 **return new** Modal(Primatives.Primitive.*createMesh*(gl, name, verts, colors));  
 }  
  
 **static** *createMesh*(gl, name, verts, colors) {  
 **let** mesh = gl.fCreateMeshVAO(name, verts, colors, gl.LINES);  
 mesh.**noCulling** = **true**;  
 mesh.**doBlending** = **true**;  
 **return** mesh;  
 }  
};

### Primatives.Point

Primatives.**Point** = **class** {  
 **static** *createModal*(gl, name, verts, colors) {  
 **return new** Modal(Primatives.Point.*createMesh*(gl, name, verts, colors));  
 }  
  
 **static** *createMesh*(gl, name, verts, colors) {  
 **let** mesh = gl.fCreateMeshVAO(name, verts, colors, gl.POINTS);  
 mesh.**noCulling** = **true**;  
 mesh.**doBlending** = **true**;  
 **return** mesh;  
 }  
};

### Primatives.GridAxis

Primatives.**GridAxis** = **class** {  
 **static** *createModal*(gl, incAxis) {  
 **return new** Modal(Primatives.GridAxis.*createMesh*(gl, incAxis));  
 }  
  
 **static** *createMesh*(gl, incAxis) {  
 **let** verts = [],  
 size = DIMENSION === 2 ? 7 : 2, div = DIMENSION === 2 ? 20 : 10, step = size / div, half = size / 2;  
 **let** p;  
 **if** (DIMENSION === 2) {  
 **for** (**let** i = 0; i <= div; i++) {  
 p = -half + (i \* step);  
 verts.push(p);  
 verts.push(half);  
 verts.push(0);  
 verts.push(0);  
  
 verts.push(p);  
 verts.push(-half);  
 verts.push(0);  
 verts.push(0);  
  
 p = half - (i \* step);  
  
 verts.push(-half);  
 verts.push(p);  
 verts.push(0);  
 verts.push(0);  
  
 verts.push(half);  
 verts.push(p);  
 verts.push(0);  
 verts.push(0);  
 }  
 } **else** {  
 **for** (**let** i = 0; i <= div; i++) {  
 p = -half + (i \* step);  
 verts.push(p);  
 verts.push(0);  
 verts.push(half);  
 verts.push(0);  
 verts.push(p);  
 verts.push(0);  
 verts.push(-half);  
 verts.push(0);  
 p = half - (i \* step);  
 verts.push(-half);  
 verts.push(0);  
 verts.push(p);  
 verts.push(0);  
 verts.push(half);  
 verts.push(0);  
 verts.push(p);  
 verts.push(0);  
 }  
 }  
  
 **if** (incAxis) {  
 verts.push(-size/2);  
 verts.push(0);  
 verts.push(0);  
 verts.push(1);  
  
 verts.push(size/2);  
 verts.push(0);  
 verts.push(0);  
 verts.push(1);  
  
 verts.push(0);  
 verts.push(-size/2);  
 verts.push(0);  
 verts.push(2);  
  
 verts.push(0);  
 verts.push(size/2);  
 verts.push(0);  
 verts.push(2);  
  
 verts.push(0);  
 verts.push(0);  
 verts.push(-size/2);  
 verts.push(3);  
  
 verts.push(0);  
 verts.push(0);  
 verts.push(size/2);  
 verts.push(3);  
 }  
  
 **let** attrColorLoc = 4,  
 strideLen,  
 mesh: **any** = {**drawMode**: gl.LINES, **vao**: gl.createVertexArray()};  
  
 mesh.**vertexComponentLen** = 4;  
 mesh.**vertexCount** = verts.**length** / mesh.vertexComponentLen;  
 strideLen = Float32Array.BYTES\_PER\_ELEMENT \* mesh.vertexComponentLen;  
 mesh.**bufVertices** = gl.createBuffer();  
 gl.bindVertexArray(mesh.**vao**);  
 gl.bindBuffer(gl.ARRAY\_BUFFER, mesh.bufVertices);  
 gl.bufferData(gl.ARRAY\_BUFFER, **new** Float32Array(verts), gl.STATIC\_DRAW);  
 gl.enableVertexAttribArray(ATTR\_POSITION\_LOC);  
 gl.enableVertexAttribArray(attrColorLoc);  
  
 gl.vertexAttribPointer(  
 ATTR\_POSITION\_LOC, 3, gl.FLOAT, **false**, strideLen, 0);  
  
 gl.vertexAttribPointer(  
 attrColorLoc, 1, gl.FLOAT  
 , **false** , strideLen, Float32Array.BYTES\_PER\_ELEMENT \* 3);  
  
 gl.bindVertexArray(**null**);  
 gl.bindBuffer(gl.ARRAY\_BUFFER, **null**);  
 gl.mMeshCache[**'grid'**] = mesh;  
 **return** mesh;  
 }  
};

### RenderLoop

*/\*\*  
 \* Контролирует перерисовку сцены.  
 \*  
 \* Tutorial on how to control FPS :: http://codetheory.in/controlling-the-frame-rate-with-requestanimationframe/  
 \* EXAMPLE:  
 \* rloop = new RenderLoop(function(dt){  
 \* console.log(rloop.fps + ' ' + dt);  
 \* },10).start();  
\*/***export class** RenderLoop {  
 **msLastFrame**;  
 **callBack**;  
 **isActive**;  
 **fps**;  
 **msFpsLimit**;  
 **run**;  
  
 **constructor**(callback, fps) {  
 **let** oThis = **this**;  
 **this**.**msLastFrame** = **null**; *// Время в миллисекундах последнего кадра.* **this**.**callBack** = callback; *// Какую функцию вызывать для каждого кадра.* **this**.**isActive** = **false**; *// Управление состоянием включения/выключения цикла рендеринга.* **this**.**fps** = 0; *// Хранит значение скорости цикла.* **if** (fps !== **undefined** && fps > 0) { *// Создать метод run, который ограничивает частоту кадров.* **this**.**msFpsLimit** = 1000 / fps; *// Считаем, сколько миллисекунд на кадр в одной секунде времени.* **this**.**run** = **function** () {  
 *// Вычислите Deltatime между кадрами и FPS в настоящее время.* **let** msCurrent = performance.now(),  
 msDelta = (msCurrent - oThis.**msLastFrame**),  
 deltaTime = msDelta / 1000.0; *// Какую долю секунды составляет Deltatime.* **if** (msDelta >= oThis.**msFpsLimit**) { *// Теперь выполним кадр как только истечет время.* oThis.**fps** = Math.floor(1 / deltaTime);  
 oThis.**msLastFrame** = msCurrent;  
 oThis.**callBack**(deltaTime);  
 }  
  
 **if** (oThis.**isActive**) window.requestAnimationFrame(oThis.**run**);  
 };  
 } **else** { *// В противном случае создайте метод run, который максимально оптимизирован.* **this**.**run** = **function** () {  
 *// Вычислите Deltatime между кадрами и FPS в настоящее время.* **let** msCurrent = performance.now(),  
 deltaTime = (msCurrent - oThis.**msLastFrame**) / 1000.0; *// Какую долю секунды составляет Deltatime.  
  
 // Теперь выполним кадр как только истечет время.* oThis.**fps** = Math.floor(1 / deltaTime); *// Время, необходимое для создания одного кадра, разделите 1 на это, чтобы получить сколько кадров за одну секунду происходит.* oThis.**msLastFrame** = msCurrent;  
  
 oThis.**callBack**(deltaTime);  
 **if** (oThis.**isActive**) window.requestAnimationFrame(oThis.**run**);  
 };  
 }  
 }  
  
 start() {  
 **this**.**isActive** = **true**;  
 **this**.**msLastFrame** = performance.now();  
 window.requestAnimationFrame(**this**.**run**);  
 **return this**;  
 }  
  
 stop() {  
 **this**.**isActive** = **false**;  
 }  
}

### Scene

**export const** ATTR\_POSITION\_NAME = **'a\_position'**;  
**export const** ATTR\_POSITION\_LOC = 0;  
**export const** ATTR\_COLOR\_NAME = **'a\_color'**;  
**export const** ATTR\_COLOR\_LOC = 1;  
**export let** DIMENSION = 3;  
  
*/\*\*  
 \* Сцена.  
 \*/***export class** Scene {  
 **canvas**;  
 **gl**;  
 **gCamera**;  
 **gCameraCtrl**;  
 **gGridShader**;  
 **gGridModal**;  
 **RLoop**;  
 **\_primitives** = [];  
 **\_points** = [];  
  
 **constructor**(canvasID, dimension) {  
 **if** (dimension === 2) {  
 DIMENSION = dimension;  
 }  
 **this**.**canvas** = document.getElementById(canvasID);  
 **const** gl = **this**.**gl** = **this**.**canvas**.getContext(**'webgl2'**);  
  
 **if** (!gl) {  
 console.error(**'WebGL context is not available.'**);  
 **return**;  
 }  
  
 gl.**mMeshCache** = []; *// Будем кешировать всё, что рисуем на сцене.  
  
 // Задаём необходимые конфигурации сцене.* gl.cullFace(gl.BACK); *// Back is also default* gl.frontFace(gl.CCW); *// Dont really need to set it, its ccw by default.* gl.enable(gl.DEPTH\_TEST); *// Shouldn't use this, use something else to add depth detection* gl.enable(gl.CULL\_FACE); *// Cull back face, so only show triangles that are created clockwise* gl.depthFunc(gl.LEQUAL); *// Near things obscure far things* gl.blendFunc(gl.SRC\_ALPHA, gl.ONE\_MINUS\_SRC\_ALPHA); *// Setup default alpha blending* gl.clearColor(.95, .95, .95, 1.0); *// Set clear color  
  
  
 // Чистим канвас от всего, что изображено.* gl.fClear = **function** () {  
 **this**.clear(**this**.COLOR\_BUFFER\_BIT | **this**.DEPTH\_BUFFER\_BIT);  
 **return this**;  
 };  
  
 *// Создаём и заполняем буффер.* gl.fCreateArrayBuffer = **function** (floatAry, isStatic = **true**) {  
 **let** buf = **this**.createBuffer();  
 **this**.bindBuffer(**this**.ARRAY\_BUFFER, buf);  
 **this**.bufferData(**this**.ARRAY\_BUFFER, floatAry, (isStatic) ? **this**.STATIC\_DRAW : **this**.DYNAMIC\_DRAW);  
 **this**.bindBuffer(**this**.ARRAY\_BUFFER, **null**);  
 **return** buf;  
 };  
  
 *// Превращаем массивы в GL буферы и заполняем VAO, которые будут предопределять буферы стандартных атрибутов шейдера.* gl.fCreateMeshVAO = **function** (name, arrVert, arrCol, drawMode = **this**.LINES) {  
 **let** res: **any** = {drawMode};  
  
 res.**vao** = **this**.createVertexArray();  
 **this**.bindVertexArray(res.**vao**);  
  
 *//.......................................................  
 // Задаём константы* **if** (arrVert) {  
 res.**bufVertices** = **this**.createBuffer();  
 res.**vertexComponentLen** = 3;  
 res.**vertexCount** = arrVert.**length** / res.vertexComponentLen;  
  
 **this**.bindBuffer(**this**.ARRAY\_BUFFER, res.bufVertices);  
 **this**.bufferData(**this**.ARRAY\_BUFFER, **new** Float32Array(arrVert), **this**.STATIC\_DRAW);  
 **this**.enableVertexAttribArray(ATTR\_POSITION\_LOC);  
 **this**.vertexAttribPointer(ATTR\_POSITION\_LOC, 3, **this**.FLOAT, **false**, 0, 0);  
 }  
  
 *//.......................................................  
 // Задаём цвета* **if** (arrCol) {  
 res.**bufColors** = **this**.createBuffer();  
  
 **this**.bindBuffer(**this**.ARRAY\_BUFFER, res.bufColors);  
 **this**.bufferData(**this**.ARRAY\_BUFFER, **new** Float32Array(arrCol), **this**.STATIC\_DRAW);  
 **this**.enableVertexAttribArray(ATTR\_COLOR\_LOC);  
 **this**.vertexAttribPointer(ATTR\_COLOR\_LOC, 4, **this**.FLOAT, **false**, 0, 0);  
 }  
  
 **this**.bindVertexArray(**null**); *// !!!* **this**.bindBuffer(**this**.ARRAY\_BUFFER, **null**);  
  
 **this**.mMeshCache[name] = res;  
 **return** res;  
 };  
  
 gl.fSetSize = **function** (w, h) {  
 **this**.**canvas**.**style**.**width** = w + **'px'**;  
 **this**.**canvas**.**style**.**height** = h + **'px'**;  
 **this**.**canvas**.**width** = w;  
 **this**.**canvas**.**height** = h;  
  
 **this**.viewport(0, 0, w, h);  
 **return this**;  
 };  
  
 gl.fFitScreen = **function** (wp = 1, hp = 1) {  
 **return this**.fSetSize(window.innerWidth \* wp, window.innerHeight \* hp);  
 };  
 }  
  
 init(incGrid, incAxis) {  
 **this**.**gl**.fFitScreen(1, 1).fClear();  
  
 **this**.**gCamera** = **new** Camera(**this**.**gl**);  
 **this**.**gCamera**.**transform**.**position**.set(0, DIMENSION === 2 ? 0 : 0.3, 3);  
 **this**.**gCameraCtrl** = **new** CameraController(**this**.**gl**, **this**.**gCamera**);  
  
 **if** (incGrid) {  
 **this**.**gGridShader** = **new** GridAxisShader(**this**.**gl**, **this**.**gCamera**.**projectionMatrix**);  
 **this**.**gGridModal** = Primatives.GridAxis.*createModal*(**this**.**gl**, incAxis);  
 }  
  
 **this**.**RLoop** = **new** RenderLoop(**this**.*onRender*, 30).start();  
 }  
  
 *onRender* = () => {  
 **this**.**gCamera**.updateViewMatrix();  
 **this**.**gl**.fClear();  
  
 **if** (**this**.**gGridShader**) {  
 **this**.**gGridShader**.activate()  
 .setCameraMatrix(**this**.**gCamera**.**viewMatrix**)  
 .renderModal(**this**.**gGridModal**.preRender());  
 }  
  
 **for** (**let** primitive **of this**.**\_primitives**) {  
 primitive.**gShader**.activate()  
 .setCameraMatrix(**this**.**gCamera**.**viewMatrix**)  
 .renderModal(primitive.**gModal**.preRender());  
 }  
 **for** (**let** point **of this**.**\_points**) {  
 point.**gShader**.activate()  
 .setCameraMatrix(**this**.**gCamera**.**viewMatrix**)  
 .renderModal(point.**gModal**.preRender());  
 }  
 };  
  
 addPrimitive(structure) {  
 **const** prim = **new** Primitive(**this**.**gl**, **this**.**gCamera**, structure);  
 **this**.**\_primitives**.push(prim);  
  
 **return** prim;  
 }  
  
 addPoint(vec, color) {  
 **this**.**\_points**.push(**new** Point(**this**.**gl**, **this**.**gCamera**, vec, color));  
 }  
}

### Point

*/\*\*  
 \* Точка.  
 \*/***class** Point {  
 **gModal**;  
 **gShader**;  
  
 **constructor**(gl, camera, vec, color) {  
 **this**.**gShader** = **new** Primitive2DShader(gl, camera.**projectionMatrix**);  
 **this**.**gModal** = Primatives.Point.*createModal*(gl, **'point'**, vec.toArray(), color.toArray());  
 }  
}

### Primitive

*/\*\*  
 \* Скелет.  
 \*/***class** Primitive {  
 **structure**;  
 **gl**;  
 **gShader**;  
 **gModal**;  
 **name**;  
 **effectors**;  
 **colors**;  
 **verts**;  
  
 **constructor**(gl, camera, structure) {  
 **this**.**structure** = structure;  
 **this**.parseStructure();  
 **this**.**gl** = gl;  
  
 **this**.**gShader** = **new** Primitive2DShader(gl, camera.**projectionMatrix**);  
 **this**.**gModal** = Primatives.Primitive.*createModal*(gl, **this**.**name**, **this**.**verts**, **this**.**colors**);  
 }  
  
 parseStructure() {  
 **this**.**name** = **this**.**structure**.*getName*();  
 **this**.**effectors** = [];  
  
 **this**.**colors** = [];  
 **this**.**verts** = [];  
  
 **for** (**let** i = 0; i < **this**.**structure**.getNumChains(); i++) {  
 **for** (**let** bone **of this**.**structure**.getChain(i).getChain()) {  
 **this**.**colors**.push(...bone.*getColour*().*toArray*());  
 **this**.**colors**.push(...bone.*getColour*().*toArray*());  
 **this**.**verts**.push(...bone.*getStartLocationAsArray*());  
 **if** (DIMENSION === 2) **this**.**verts**.push(0);  
 **this**.**verts**.push(...bone.*getEndLocationAsArray*());  
 **if** (DIMENSION === 2) **this**.**verts**.push(0);  
 }  
 **this**.**effectors**.push(**this**.**structure**.getChain(i));  
 }  
 }  
  
 moveTo(index, vec) {  
 **for** (**let** i = 0; i < **this**.**structure**.getNumChains(); i++) {  
 **this**.**structure**.getChain(i).setEmbeddedTargetMode(index !== i);  
 **if** (index !== i) {  
 **this**.**structure**.getChain(i).updateEmbeddedTarget(**this**.**structure**.getChain(i).getEffectorLocation());  
 }  
 }  
 **this**.**structure**.solveForTarget(vec);  
 **this**.parseStructure();  
 **this**.**gModal** = Primatives.Primitive.*createModal*(**this**.**gl**, **this**.**name**, **this**.**verts**, **this**.**colors**);  
 }  
}

### Shader

*/\*\*  
 \* Шейдер.  
 \*/***class** Shader {  
 **program**;  
 **gl**;  
 **attribLoc**;  
 **uniformLoc**;  
  
 **constructor**(gl, vertShaderSrc, fragShaderSrc) {  
 **this**.**program** = ShaderUtil.*createProgramFromText*(gl, vertShaderSrc, fragShaderSrc);  
  
 **if** (**this**.**program** !== **null**) {  
 **this**.**gl** = gl;  
 gl.useProgram(**this**.**program**);  
 **this**.**attribLoc** = ShaderUtil.*getStandardAttribLocations*(gl, **this**.**program**);  
 **this**.**uniformLoc** = ShaderUtil.*getStandardUniformLocations*(gl, **this**.**program**);  
 }  
 }  
  
 activate() {  
 **this**.**gl**.useProgram(**this**.**program**);  
 **return this**;  
 }  
  
 deactivate() {  
 **this**.**gl**.useProgram(**null**);  
 **return this**;  
 }  
  
 setPerspective(matData) {  
 **this**.**gl**.uniformMatrix4fv(**this**.**uniformLoc**.**perspective**, **false**, matData);  
 **return this**;  
 }  
  
 setModalMatrix(matData) {  
 **this**.**gl**.uniformMatrix4fv(**this**.**uniformLoc**.**modalMatrix**, **false**, matData);  
 **return this**;  
 }  
  
 setCameraMatrix(matData) {  
 **this**.**gl**.uniformMatrix4fv(**this**.**uniformLoc**.**cameraMatrix**, **false**, matData);  
 **return this**;  
 }  
  
 dispose() {  
 **if** (**this**.**gl**.getParameter(**this**.**gl**.CURRENT\_PROGRAM) === **this**.**program**) **this**.**gl**.useProgram(**null**);  
 **this**.**gl**.deleteProgram(**this**.**program**);  
 }  
  
 preRender() {  
 }  
  
 renderModal(modal) {  
 **this**.setModalMatrix(modal.**transform**.getViewMatrix());  
 **this**.**gl**.bindVertexArray(modal.**mesh**.**vao**);  
  
 **if** (modal.**mesh**.noCulling) **this**.**gl**.disable(**this**.**gl**.CULL\_FACE);  
 **if** (modal.**mesh**.doBlending) **this**.**gl**.enable(**this**.**gl**.BLEND);  
  
 **if** (modal.**mesh**.indexCount) **this**.**gl**.drawElements(modal.**mesh**.**drawMode**, modal.**mesh**.indexCount, **this**.**gl**.UNSIGNED\_SHORT, 0);  
 **else this**.**gl**.drawArrays(modal.**mesh**.**drawMode**, 0, modal.**mesh**.vertexCount);  
  
 **this**.**gl**.bindVertexArray(**null**);  
 **if** (modal.**mesh**.noCulling) **this**.**gl**.enable(**this**.**gl**.CULL\_FACE);  
 **if** (modal.**mesh**.doBlending) **this**.**gl**.disable(**this**.**gl**.BLEND);  
  
 **return this**;  
 }  
}

### ShaderUtil

*/\*\*  
 \* Вспомогательные методы для работы с шейдерами.  
 \*/***class** ShaderUtil {  
 **static** *createShader*(gl, src, type) {  
 **let** shader = gl.createShader(type);  
 gl.shaderSource(shader, src);  
 gl.compileShader(shader);  
  
 **if** (!gl.getShaderParameter(shader, gl.COMPILE\_STATUS)) {  
 console.error(**'Error compiling shader : '** + src, gl.getShaderInfoLog(shader));  
 gl.deleteShader(shader);  
 **return null**;  
 }  
  
 **return** shader;  
 }  
  
 **static** *createProgram*(gl, vShader, fShader, doValidate) {  
 **let** prog = gl.createProgram();  
 gl.attachShader(prog, vShader);  
 gl.attachShader(prog, fShader);  
  
 gl.bindAttribLocation(prog, ATTR\_POSITION\_LOC, ATTR\_POSITION\_NAME);  
 gl.bindAttribLocation(prog, ATTR\_COLOR\_LOC, ATTR\_COLOR\_NAME);  
  
 gl.linkProgram(prog);  
  
 **if** (!gl.getProgramParameter(prog, gl.LINK\_STATUS)) {  
 console.error(**'Error creating shader program.'**, gl.getProgramInfoLog(prog));  
 gl.deleteProgram(prog);  
 **return null**;  
 }  
  
 **if** (doValidate) {  
 gl.validateProgram(prog);  
 **if** (!gl.getProgramParameter(prog, gl.VALIDATE\_STATUS)) {  
 console.error(**'Error validating program'**, gl.getProgramInfoLog(prog));  
 gl.deleteProgram(prog);  
 **return null**;  
 }  
 }  
  
 gl.detachShader(prog, vShader);  
 gl.detachShader(prog, fShader);  
 gl.deleteShader(fShader);  
 gl.deleteShader(vShader);  
  
 **return** prog;  
 }  
  
 **static** *createProgramFromText*(gl, vShaderTxt, fShaderTxt) {  
 **let** vShader = ShaderUtil.*createShader*(gl, vShaderTxt, gl.VERTEX\_SHADER);  
 **if** (!vShader) **return null**;  
 **let** fShader = ShaderUtil.*createShader*(gl, fShaderTxt, gl.FRAGMENT\_SHADER);  
 **if** (!fShader) {  
 gl.deleteShader(vShader);  
 **return null**;  
 }  
  
 **return** ShaderUtil.*createProgram*(gl, vShader, fShader, **true**);  
 }  
  
 **static** *getStandardAttribLocations*(gl, program) {  
 **return** {  
 **position**: gl.getAttribLocation(program, ATTR\_POSITION\_NAME),  
 **color**: gl.getAttribLocation(program, ATTR\_COLOR\_NAME),  
 };  
 }  
  
 **static** *getStandardUniformLocations*(gl, program) {  
 **return** {  
 **perspective**: gl.getUniformLocation(program, **'uPMatrix'**),  
 **modalMatrix**: gl.getUniformLocation(program, **'uMVMatrix'**),  
 **cameraMatrix**: gl.getUniformLocation(program, **'uCameraMatrix'**),  
 **mainTexture**: gl.getUniformLocation(program, **'uMainTex'**),  
 };  
 }  
}

### GridAxisShader

*/\*\*  
 \* Шейдер грида.  
 \*/***export class** GridAxisShader **extends** Shader {  
 **constructor**(gl, pMatrix) {  
 **const** vertSrc = **'#version 300 es\n'** +  
 **'in vec3 a\_position;'** +  
 **'layout(location=4) in float a\_color;'** +  
 **'uniform mat4 uPMatrix;'** +  
 **'uniform mat4 uMVMatrix;'** +  
 **'uniform mat4 uCameraMatrix;'** +  
 **'uniform vec3 uColor[4];'** +  
 **'out lowp vec4 color;'** +  
 **'void main(void){'** +  
 **'color = vec4(uColor[ int(a\_color) ],1.0);'** +  
 **'gl\_Position = uPMatrix \* uCameraMatrix \* uMVMatrix \* vec4(a\_position, 1.0);'** +  
 **'}'**;  
 **const** fragSrc = **'#version 300 es\n'** +  
 **'precision mediump float;'** +  
 **'in vec4 color;'** +  
 **'out vec4 finalColor;'** +  
 **'void main(void){ finalColor = color; }'**;  
  
 **super**(gl, vertSrc, fragSrc);  
  
 **this**.setPerspective(pMatrix);  
  
 **const** uColor = gl.getUniformLocation(**this**.**program**, **'uColor'**);  
 gl.uniform3fv(uColor, **new** Float32Array([0.8, 0.8, 0.8, 1, 0, 0, 0, 1, 0, 0, 0, 1]));  
  
 gl.useProgram(**null**);  
 }  
}

### Primitive2DShader

*/\*\*  
 \* Шейдер для точки и скелета.  
 \*/***export class** Primitive2DShader **extends** Shader {  
 **constructor**(gl, pMatrix) {  
 **const** vertSrc = **`#version 300 es  
 in vec3 a\_position;  
 in vec4 a\_color;  
  
 uniform mat4 uPMatrix;  
 uniform mat4 uMVMatrix;  
 uniform mat4 uCameraMatrix;  
  
 out vec4 color;  
  
 void main(void) {  
 gl\_PointSize = 4.0;  
 color = a\_color;  
 gl\_Position = uPMatrix \* uCameraMatrix \* uMVMatrix \* vec4(a\_position, 1.0);   
 }`**;  
 **const** fragSrc = **`#version 300 es  
 precision mediump float;  
 in vec4 color;  
  
 out vec4 finalColor;  
   
 void main(void) {  
 finalColor = color;  
 }`**;  
 **super**(gl, vertSrc, fragSrc);  
  
 **this**.setPerspective(pMatrix);  
 gl.useProgram(**null**);  
 }  
}

### Transform

*/\*\*  
 \* Положение в пространстве сцены.  
 \*/***export class** Transform {  
 **position**: Vec3f;  
 **scale**: Vec3f;  
 **rotation**: Vec3f;  
 **matView**: Mat4f;  
 **matNormal**: Float32Array;  
 **forward**: Float32Array;  
 **up**: Float32Array;  
 **right**: Float32Array;  
  
 **constructor**() {  
 **this**.**position** = **new** Vec3f(0, 0, 0);  
 **this**.**scale** = **new** Vec3f(1, 1, 1);  
 **this**.**rotation** = **new** Vec3f(0, 0, 0);  
 **this**.**matView** = **new** Mat4f();  
 **this**.**matNormal** = **new** Float32Array(9);  
  
 **this**.**forward** = **new** Float32Array(4);  
 **this**.**up** = **new** Float32Array(4);  
 **this**.**right** = **new** Float32Array(4);  
 }  
  
 updateMatrix() {  
 **this**.**matView**.reset()  
 .vtranslate(**this**.**position**)  
 .rotateX(**this**.**rotation**.**x** \* Utils.*DEGS\_TO\_RADS*)  
 .rotateZ(**this**.**rotation**.**z** \* Utils.*DEGS\_TO\_RADS*)  
 .rotateY(**this**.**rotation**.**y** \* Utils.*DEGS\_TO\_RADS*)  
 .vscale(**this**.**scale**);  
  
 Mat4f.*normalMat3*(**this**.**matNormal**, **this**.**matView**.**raw**);  
  
 Mat4f.*transformVec4*(**this**.**forward**, [0, 0, 1, 0], **this**.**matView**.**raw**);  
 Mat4f.*transformVec4*(**this**.**up**, [0, 1, 0, 0], **this**.**matView**.**raw**);  
 Mat4f.*transformVec4*(**this**.**right**, [1, 0, 0, 0], **this**.**matView**.**raw**);  
  
 **return this**.**matView**.**raw**;  
 }  
  
 updateDirection() {  
 Mat4f.*transformVec4*(**this**.**forward**, [0, 0, 1, 0], **this**.**matView**.**raw**);  
 Mat4f.*transformVec4*(**this**.**up**, [0, 1, 0, 0], **this**.**matView**.**raw**);  
 Mat4f.*transformVec4*(**this**.**right**, [1, 0, 0, 0], **this**.**matView**.**raw**);  
 **return this**;  
 }  
  
 getViewMatrix() {  
 **return this**.**matView**.**raw**;  
 }  
  
 reset() {  
 **this**.**position**.set(0, 0, 0);  
 **this**.**scale**.set(1, 1, 1);  
 **this**.**rotation**.set(0, 0, 0);  
 }  
}

## Вспомогательный модуль

### Colour4f

*/\*\*  
 \* Цвет в формате RGBA.  
 \* <p>  
 \* Представлен четырьмя r, g, b, a свойствами типа Number.  
 \* Задать значения данных свойств можно присваиванием напрямую, через конструктор или  
 \* через функции сеттеры.  
 \* Допустимые значения для каждой переменной - число [0..1].  
 \* Использование значений вне этого диапазона может привести к неопределенному поведению.  
 \*/***export class** Colour4f {  
 *// ---------- Properties ----------  
  
 /\*\* Минимально допустимое значение для компоненты цвета. \*/* **static** *MIN\_COMPONENT\_VALUE* = 0;  
  
 */\*\* Максимально допустимое значение для компоненты цвета. \*/* **static** *MAX\_COMPONENT\_VALUE* = 1;  
  
 */\*\* Red компонента. \*/* **r**: **number** = 1;  
  
 */\*\* Green компонента. \*/* **g**: **number** = 1;  
  
 */\*\* Blue компонента. \*/* **b**: **number** = 1;  
  
 */\*\* Alpha компонента (компонента непрозрачности).  
 \* \*/* **a**: **number** = 1;  
  
 *// ---------- Constructor ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает один из двух возможных сочетаний параметров:<ul>  
 \* <li>  
 \* Без параметров - проставляет значения по умолчанию всем полям.  
 \* </li>  
 \* <li>  
 \* Принимает массив из 4 значений для компонент по порядку: [red, green, blue, alpha].  
 \*  
 \** ***@param*** *source {number[]} Значения компонент цвета.  
 \* </li>  
 \* </ul>  
 \*  
 \* Все значения должны быть в пределах [0..1].  
 \*/* **constructor**(source?: **number**[]) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **break**;  
 **case** 1:  
 **if** (source **instanceof** Array && source.**length** == 4) {  
 **this**.**r** = Colour4f.*\_clamp*(source[0]);  
 **this**.**g** = Colour4f.*\_clamp*(source[1]);  
 **this**.**b** = Colour4f.*\_clamp*(source[2]);  
 **this**.**a** = Colour4f.*\_clamp*(source[3]);  
 } **else** {  
 **throw new** Error(**'Colour source array size must be precisely 4 elements.'**);  
 }  
 **break**;  
 **default**:  
 **throw new** Error(**'Colour constructor was invoked incorrectly.'**);  
 }  
 }  
  
 *// ---------- Public Methods ----------  
  
 /\*\*  
 \* Задёт значения компонент.  
 \* <p>  
 \* Если значения не подходят по диапазону, то они заменяются на граничные.  
 \*  
 \** ***@param*** *source Массив из 4 значений для компонент по порядку: [red, green, blue, alpha].  
 \*/* set(source: **number**[]): **void** {  
 **this**.**r** = Colour4f.*\_clamp*(source[0]);  
 **this**.**g** = Colour4f.*\_clamp*(source[1]);  
 **this**.**b** = Colour4f.*\_clamp*(source[2]);  
 **this**.**a** = Colour4f.*\_clamp*(source[3]);  
 };  
  
 */\*\*  
 \* Добавляет заданные значения к RGB компонентам и возвращает изменненный цвет для формирования цепочки.  
 \* <p>  
 \* Если при добавленни значений происходит выход за границы диапазона,  
 \* то результат заменяется на ближайшее граничное.  
 \*  
 \** ***@param*** *red Red компонента для добавления.  
 \** ***@param*** *green Green компонента для добавления.  
 \** ***@param*** *blue Blue компонента для добавления.  
 \** ***@return*** *This модифицированный цвет.  
 \*/* addRGB (red: **number**, green: **number**, blue: **number**): Colour4f {  
 **this**.**r** = Colour4f.*\_clamp*(**this**.**r** + red);  
 **this**.**g** = Colour4f.*\_clamp*(**this**.**g** + green);  
 **this**.**b** = Colour4f.*\_clamp*(**this**.**b** + blue);  
 **return this**;  
 };  
  
 */\*\*  
 \* Вычитает заданные значения из RGB компонент и возвращает изменненный цвет для формирования цепочки.  
 \* <p>  
 \* Если при вычитании значений происходит выход за границы диапазона,  
 \* то результат заменяется на ближайшее граничное.  
 \*  
 \** ***@param*** *red Red компонента для вычитания.  
 \** ***@param*** *green Green компонента для вычитания.  
 \** ***@param*** *blue Blue компонента для вычитания.  
 \** ***@return*** *This модифицированный цвет.  
 \*/* subtractRGB (red: **number**, green: **number**, blue: **number**): Colour4f {  
 **this**.**r** = Colour4f.*\_clamp*(**this**.**r** - red);  
 **this**.**g** = Colour4f.*\_clamp*(**this**.**g** - green);  
 **this**.**b** = Colour4f.*\_clamp*(**this**.**b** - blue);  
 **return this**;  
 };  
  
 */\*\*  
 \* Осветляет RGB компоненты данного цвета на заданное количество единиц.  
 \* <p>  
 \* Результирующий цвет приводится к диапазону [0..1] и возвращается для создания цепочки.  
 \*  
 \** ***@param*** *amount Значение, которое необходимо прибавить к RGB компонентам текущего цвета.  
 \** ***@return*** *Текущий цвет после модицикации.  
 \*/* lighten (amount: **number**): Colour4f {  
 **return this**.addRGB(amount, amount, amount);  
 }  
  
 */\*\*  
 \* Затемняет RGB компоненты данного цвета на заданное количество единиц.  
 \* <p>  
 \* Результирующий цвет приводится к диапазону [0..1] и возвращается для создания цепочки.  
 \*  
 \** ***@param*** *amount Значение, которое необходимо отнять от RGB компонент текущего цвета.  
 \** ***@return*** *Текущий цвет после модицикации.  
 \*/* darken (amount: **number**): Colour4f {  
 **return this**.subtractRGB(amount, amount, amount);  
 }  
  
 */\*\*  
 \* Возвращает массив компонент данного цвета в порядке RGBA.  
 \*  
 \** ***@return*** *Данный цвет в виде числового массива.  
 \*/  
 toArray* = (): **number**[] => [**this**.**r**, **this**.**g**, **this**.**b**, **this**.**a**];  
  
 *// ---------- Private Methods ----------* **private static** *\_clamp*(componentValue: **number**): **number** {  
 **if** (componentValue > Colour4f.*MAX\_COMPONENT\_VALUE*) { **return** Colour4f.*MAX\_COMPONENT\_VALUE*; }  
 **else if** (componentValue < Colour4f.*MIN\_COMPONENT\_VALUE*) { **return** Colour4f.*MIN\_COMPONENT\_VALUE*; }  
 **else** { **return** componentValue; }  
 }  
}

### Mat3f

**import** {Vec3f} **from './Vec3f'**;  
**import** {Utils} **from './Utils'**;  
  
*/\*\*  
 \* Числовая матрица 3x3.  
 \* <p>  
 \* Элементы данной матрицы:  
 \* m00 m10 m20  
 \* m01 m11 m21  
 \* m02 m12 m22  
 \* <p>  
 \* Основной элемент матрицы - столбец.  
 \*  
 \*/***export class** Mat3f {  
 *// Первый вектор-столбец - положительно направление оси X.* **m00**: **number** = 0;  
 **m01**: **number** = 0;  
 **m02**: **number** = 0;  
  
 *// Второй вектор-столбец - положительно направление оси Y.* **m10**: **number** = 0;  
 **m11**: **number** = 0;  
 **m12**: **number** = 0;  
  
 *// Третий вектор-столбец - положительно направление оси Z.* **m20**: **number** = 0;  
 **m21**: **number** = 0;  
 **m22**: **number** = 0;  
  
 */\*\*  
 \* Конструктор.  
 \* Принимает 9 параметров или ноль.  
 \*  
 \** ***@param*** *m00 Первый компонент по оси X.  
 \** ***@param*** *m01 Второй компонент по оси X.  
 \** ***@param*** *m02 Третий компонент по оси X.  
 \** ***@param*** *m10 Первый компонент по оси Y.  
 \** ***@param*** *m11 Второй компонент по оси Y.  
 \** ***@param*** *m12 Третий компонент по оси Y.  
 \** ***@param*** *m20 Первый компонент по оси Z.  
 \** ***@param*** *m21 Второй компонент по оси Z.  
 \** ***@param*** *m22 Третий компонент по оси Z.  
 \*/* **constructor**(m00?: **number**, m01?: **number**, m02?: **number**, m10?: **number**, m11?: **number**, m12?: **number**, m20?: **number**, m21?: **number**, m22?: **number**) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **return**;  
 **case** 9:  
 **this**.**m00** = m00;  
 **this**.**m01** = m01;  
 **this**.**m02** = m02;  
  
 **this**.**m10** = m10;  
 **this**.**m11** = m11;  
 **this**.**m12** = m12;  
  
 **this**.**m20** = m20;  
 **this**.**m21** = m21;  
 **this**.**m22** = m22;  
 **return**;  
 **default**:  
 **throw** Error(**'Invalid Mat3f constructor params.'**)  
 }  
 }  
  
 */\*\* Зануляет все числа в матрице. \*/* zero(): **void** { **this**.**m00** = **this**.**m01** = **this**.**m02** = **this**.**m10** = **this**.**m11** = **this**.**m12** = **this**.**m20** = **this**.**m21** = **this**.**m22** = 0.0; }  
  
 */\*\* Превращает данную матрицу в единичную. \*/* setIdentity(): **void** {  
 **this**.**m00** = **this**.**m11** = **this**.**m22** = 1.0;  
 **this**.**m01** = **this**.**m02** = **this**.**m10** = **this**.**m12** = **this**.**m20** = **this**.**m21** = 0.0;  
 }  
  
 */\*\*  
 \* Возвращает новую матрицу, транспонированную версию матрицы-параметра.  
 \*  
 \** ***@param*** *m Матрицу, которую хотят транспонировать.  
 \** ***@return*** *Транспонированная версия матрицы-параметра.  
 \*/* **static** *transpose* = (m: Mat3f): Mat3f => **new** Mat3f(m.**m00**, m.**m10**, m.**m20**, m.**m01**, m.**m11**, m.**m21**, m.**m02**, m.**m12**, m.**m22**);  
  
 */\*\*  
 \* Задаёт матрицу вращения.  
 \*  
 \** ***@param*** *referenceDirection Вектор используемый в качестве оси Z.  
 \** ***@return*** *Матрица вращения.  
 \*/* **static** *createRotationMatrix*(referenceDirection: Vec3f): Mat3f {  
 **const** rotMat = **new** Mat3f();  
  
 **if** (referenceDirection.**y** == 1.0) {  
 referenceDirection.**y** -= 0.0001;  
 referenceDirection.normalise();  
 }  
  
 rotMat.setZBasis( referenceDirection );  
 rotMat.setXBasis( Vec3f.*crossProduct*( referenceDirection, **new** Vec3f(0.0, 1.0, 0.0) ).normalised() );  
 rotMat.setYBasis( Vec3f.*crossProduct*( rotMat.*getXBasis*(), rotMat.*getZBasis*() ).normalised() );  
  
 **return** rotMat;  
 }  
  
 */\*\*  
 \* Состоит ли матрица из трёх ортогональных векторов.  
 \*  
 \** ***@return*** *Ортогональна ли матрица.  
 \*/* isOrthogonal(): **boolean** {  
 **const** xCrossYDot = Vec3f.*dotProduct*(**this**.*getXBasis*(), **this**.*getYBasis*());  
 **const** xCrossZDot = Vec3f.*dotProduct*(**this**.*getXBasis*(), **this**.*getZBasis*());  
 **const** yCrossZDot = Vec3f.*dotProduct*(**this**.*getYBasis*(), **this**.*getZBasis*());  
  
 **return** (Utils.*approximatelyEquals*(xCrossYDot, 0.0, 0.01) &&  
 Utils.*approximatelyEquals*(xCrossZDot, 0.0, 0.01) &&  
 Utils.*approximatelyEquals*(yCrossZDot, 0.0, 0.01));  
 }  
  
 */\*\*  
 \* Умножает матрицу на вектор обычным матричным умножением.  
 \*  
 \** ***@param*** *source Вектор.  
 \** ***@return*** *\*/* times(source: Vec3f): Vec3f {  
 **return new** Vec3f(**this**.**m00** \* source.**x** + **this**.**m10** \* source.**y** + **this**.**m20** \* source.**z**,  
 **this**.**m01** \* source.**x** + **this**.**m11** \* source.**y** + **this**.**m21** \* source.**z**,  
 **this**.**m02** \* source.**x** + **this**.**m12** \* source.**y** + **this**.**m22** \* source.**z**);  
 }  
  
 */\*\*  
 \* Вычисляет определитель матрицы.  
 \*  
 \** ***@return*** *Определитель матрицы.  
 \*/* determinant() {  
 **return this**.**m20** \* **this**.**m01** \* **this**.**m12** - **this**.**m20** \* **this**.**m02** \* **this**.**m11** - **this**.**m10** \* **this**.**m01** \* **this**.**m22** + **this**.**m10** \* **this**.**m02** \* **this**.**m21** + **this**.**m00** \* **this**.**m11** \* **this**.**m22** - **this**.**m00** \* **this**.**m12** \* **this**.**m21**;  
 }  
  
 */\*\*  
 \* Возвращает обратную матрицу для матрицы-параметра.  
 \*  
 \** ***@param*** *m Исходная матрица.  
 \** ***@return*** *Матрица обратная исходной.  
 \*/* **static** *inverse*(m: Mat3f): Mat3f {  
 **const** d = m.determinant();  
  
 **const** temp = **new** Mat3f();  
  
 temp.**m00** = (m.**m11** \* m.**m22** - m.**m12** \* m.**m21**) / d;  
 temp.**m01** = -(m.**m01** \* m.**m22** - m.**m02** \* m.**m21**) / d;  
 temp.**m02** = (m.**m01** \* m.**m12** - m.**m02** \* m.**m11**) / d;  
 temp.**m10** = -(-m.**m20** \* m.**m12** + m.**m10** \* m.**m22**) / d;  
 temp.**m11** = (-m.**m20** \* m.**m02** + m.**m00** \* m.**m22**) / d;  
 temp.**m12** = -(-m.**m10** \* m.**m02** + m.**m00** \* m.**m12**) / d;  
 temp.**m20** = (-m.**m20** \* m.**m11** + m.**m10** \* m.**m21**) / d;  
 temp.**m21** = -(-m.**m20** \* m.**m01** + m.**m00** \* m.**m21**) / d;  
 temp.**m22** = (-m.**m10** \* m.**m02** + m.**m00** \* m.**m11**) / d;  
  
 **return** temp;  
 }  
  
 */\*\*  
 \* Поворачивает эту матрицу на угол-параметр относительно оси-параметра.  
 \* Не модифицирует исходную матрицу.  
 \*  
 \** ***@param*** *angleRads Угол в радианах.  
 \** ***@param*** *rotationAxis Ось.  
 \** ***@return*** *\* \*/* rotateRads(rotationAxis: Vec3f, angleRads: **number**): Mat3f {  
 **const** dest = **new** Mat3f();  
  
 **const** sin = Math.sin(angleRads);  
 **const** cos = Math.cos(angleRads);  
 **const** oneMinusCos = 1.0 - cos;  
  
 **const** xy = rotationAxis.**x** \* rotationAxis.**y**;  
 **const** yz = rotationAxis.**y** \* rotationAxis.**z**;  
 **const** xz = rotationAxis.**x** \* rotationAxis.**z**;  
 **const** xs = rotationAxis.**x** \* sin;  
 **const** ys = rotationAxis.**y** \* sin;  
 **const** zs = rotationAxis.**z** \* sin;  
  
 **const** f00 = rotationAxis.**x** \* rotationAxis.**x** \* oneMinusCos + cos;  
 **const** f01 = xy \* oneMinusCos + zs;  
 **const** f02 = xz \* oneMinusCos - ys;  
  
 **const** f10 = xy \* oneMinusCos - zs;  
 **const** f11 = rotationAxis.**y** \* rotationAxis.**y** \* oneMinusCos + cos;  
 **const** f12 = yz \* oneMinusCos + xs;  
  
 **const** f20 = xz \* oneMinusCos + ys;  
 **const** f21 = yz \* oneMinusCos - xs;  
 **const** f22 = rotationAxis.**z** \* rotationAxis.**z** \* oneMinusCos + cos;  
  
 **const** t00 = **this**.**m00** \* f00 + **this**.**m10** \* f01 + **this**.**m20** \* f02;  
 **const** t01 = **this**.**m01** \* f00 + **this**.**m11** \* f01 + **this**.**m21** \* f02;  
 **const** t02 = **this**.**m02** \* f00 + **this**.**m12** \* f01 + **this**.**m22** \* f02;  
  
 **const** t10 = **this**.**m00** \* f10 + **this**.**m10** \* f11 + **this**.**m20** \* f12;  
 **const** t11 = **this**.**m01** \* f10 + **this**.**m11** \* f11 + **this**.**m21** \* f12;  
 **const** t12 = **this**.**m02** \* f10 + **this**.**m12** \* f11 + **this**.**m22** \* f12;  
  
 dest.**m20** = **this**.**m00** \* f20 + **this**.**m10** \* f21 + **this**.**m20** \* f22;  
 dest.**m21** = **this**.**m01** \* f20 + **this**.**m11** \* f21 + **this**.**m21** \* f22;  
 dest.**m22** = **this**.**m02** \* f20 + **this**.**m12** \* f21 + **this**.**m22** \* f22;  
  
 dest.**m00** = t00;  
 dest.**m01** = t01;  
 dest.**m02** = t02;  
  
 dest.**m10** = t10;  
 dest.**m11** = t11;  
 dest.**m12** = t12;  
  
 **return** dest;  
 }  
  
 */\*\*  
 \* Поворачивает эту матрицу на угол-параметр относительно оси-параметра.  
 \* Не модифицирует исходную матрицу.  
 \*  
 \** ***@param*** *angleDegs Угол в градусах.  
 \** ***@param*** *localAxis Ось.  
 \** ***@return*** *\* \*/  
 rotateDegs* = (angleDegs: **number**, localAxis: Vec3f): Mat3f => **this**.rotateRads(localAxis, angleDegs \* Utils.*DEGS\_TO\_RADS*);  
  
 */\*\*  
 \* Задаёт ось X матрице.  
 \*  
 \** ***@param*** *v Вектор, значения которого будут использоваться в качестве оси матрицы.  
 \*/* setXBasis(v: Vec3f): **void** { **this**.**m00** = v.**x**; **this**.**m01** = v.**y**; **this**.**m02** = v.**z**; }  
  
 */\*\*  
 \* Возвращает значения матрицы по оси X.  
 \*  
 \** ***@return*** *\*\*/  
 getXBasis* = (): Vec3f => **new** Vec3f(**this**.**m00**, **this**.**m01**, **this**.**m02**);  
  
 */\*\*  
 \* Задаёт ось Y матрице.  
 \*  
 \** ***@param*** *v Вектор, значения которого будут использоваться в качестве оси матрицы.  
 \*/* setYBasis(v: Vec3f): **void** { **this**.**m10** = v.**x**; **this**.**m11** = v.**y**; **this**.**m12** = v.**z**; }  
  
 */\*\*  
 \* Возвращает значения матрицы по оси Y.  
 \*  
 \** ***@return*** *\*\*/  
 getYBasis* = (): Vec3f => **new** Vec3f(**this**.**m10**, **this**.**m11**, **this**.**m12**);  
  
 */\*\*  
 \* Задаёт ось Z матрице.  
 \*  
 \** ***@param*** *v Вектор, значения которого будут использоваться в качестве оси матрицы.  
 \*/* setZBasis(v: Vec3f): **void** { **this**.**m20** = v.**x**; **this**.**m21** = v.**y**; **this**.**m22** = v.**z**; }  
  
 */\*\*  
 \* Возвращает значения матрицы по оси Z.  
 \*  
 \** ***@return*** *\*\*/  
 getZBasis* = (): Vec3f => **new** Vec3f(**this**.**m20**, **this**.**m21**, **this**.**m22**);  
  
 */\*\*  
 \* Возвращает матрицу в виде массива из девяти чисел.  
 \*  
 \** ***@return*** *Матрица как массив из 9 чисел.  
 \*/  
 toArray* = (): **number**[] => [**this**.**m00**, **this**.**m01**, **this**.**m02**, **this**.**m10**, **this**.**m11**, **this**.**m12**, **this**.**m20**, **this**.**m21**, **this**.**m22**];  
  
 **static** *identity*() {  
 **let** a = **new** Float32Array(9);  
 a[0] = a[4] = a[8] = 1;  
 **return** a;  
 }  
}

### Mat4f

*/\*\*  
 \* Матрица 4x4.  
 \*  
 \* Класс, используемый для отрисовки.  
 \*/***export class** Mat4f {  
 **raw**;  
  
 **constructor**() {  
 **this**.**raw** = Mat4f.*identity*();  
 }  
  
 vtranslate(v) {  
 Mat4f.*translate*(**this**.**raw**, v.**x**, v.**y**, v.**z**);  
 **return this**;  
 }  
  
 rotateY(rad) {  
 Mat4f.*rotateY*(**this**.**raw**, rad);  
 **return this**;  
 }  
  
 rotateX(rad) {  
 Mat4f.*rotateX*(**this**.**raw**, rad);  
 **return this**;  
 }  
  
 rotateZ(rad) {  
 Mat4f.*rotateZ*(**this**.**raw**, rad);  
 **return this**;  
 }  
  
 vscale(vec3) {  
 Mat4f.*scale*(**this**.**raw**, vec3.**x**, vec3.**y**, vec3.**z**);  
 **return this**;  
 }  
  
 reset() {  
 **for** (**let** i = 0; i < **this**.**raw**.**length**; i++) **this**.**raw**[i] = (i % 5 === 0) ? 1 : 0;  
 **return this**;  
 }  
  
 **static** *identity*() {  
 **let** a = **new** Float32Array(16);  
 a[0] = a[5] = a[10] = a[15] = 1;  
 **return** a;  
 }  
  
 **static** *perspective*(out, fovy, aspect, near, far) {  
 **let** f = 1.0 / Math.tan(fovy / 2),  
 nf = 1 / (near - far);  
 out[0] = f / aspect;  
 out[1] = 0;  
 out[2] = 0;  
 out[3] = 0;  
 out[4] = 0;  
 out[5] = f;  
 out[6] = 0;  
 out[7] = 0;  
 out[8] = 0;  
 out[9] = 0;  
 out[10] = (far + near) \* nf;  
 out[11] = -1;  
 out[12] = 0;  
 out[13] = 0;  
 out[14] = (2 \* far \* near) \* nf;  
 out[15] = 0;  
 }  
  
 **static** *normalMat3*(out, a) {  
 **let** a00 = a[0], a01 = a[1], a02 = a[2], a03 = a[3],  
 a10 = a[4], a11 = a[5], a12 = a[6], a13 = a[7],  
 a20 = a[8], a21 = a[9], a22 = a[10], a23 = a[11],  
 a30 = a[12], a31 = a[13], a32 = a[14], a33 = a[15],  
  
 b00 = a00 \* a11 - a01 \* a10,  
 b01 = a00 \* a12 - a02 \* a10,  
 b02 = a00 \* a13 - a03 \* a10,  
 b03 = a01 \* a12 - a02 \* a11,  
 b04 = a01 \* a13 - a03 \* a11,  
 b05 = a02 \* a13 - a03 \* a12,  
 b06 = a20 \* a31 - a21 \* a30,  
 b07 = a20 \* a32 - a22 \* a30,  
 b08 = a20 \* a33 - a23 \* a30,  
 b09 = a21 \* a32 - a22 \* a31,  
 b10 = a21 \* a33 - a23 \* a31,  
 b11 = a22 \* a33 - a23 \* a32,  
  
  
 det = b00 \* b11 - b01 \* b10 + b02 \* b09 + b03 \* b08 - b04 \* b07 + b05 \* b06;  
  
 **if** (!det) **return null**;  
  
 det = 1.0 / det;  
  
 out[0] = (a11 \* b11 - a12 \* b10 + a13 \* b09) \* det;  
 out[1] = (a12 \* b08 - a10 \* b11 - a13 \* b07) \* det;  
 out[2] = (a10 \* b10 - a11 \* b08 + a13 \* b06) \* det;  
  
 out[3] = (a02 \* b10 - a01 \* b11 - a03 \* b09) \* det;  
 out[4] = (a00 \* b11 - a02 \* b08 + a03 \* b07) \* det;  
 out[5] = (a01 \* b08 - a00 \* b10 - a03 \* b06) \* det;  
  
 out[6] = (a31 \* b05 - a32 \* b04 + a33 \* b03) \* det;  
 out[7] = (a32 \* b02 - a30 \* b05 - a33 \* b01) \* det;  
 out[8] = (a30 \* b04 - a31 \* b02 + a33 \* b00) \* det;  
 **return** out;  
 }  
  
 **static** *transformVec4*(out, v, m) {  
 out[0] = m[0] \* v[0] + m[4] \* v[1] + m[8] \* v[2] + m[12] \* v[3];  
 out[1] = m[1] \* v[0] + m[5] \* v[1] + m[9] \* v[2] + m[13] \* v[3];  
 out[2] = m[2] \* v[0] + m[6] \* v[1] + m[10] \* v[2] + m[14] \* v[3];  
 out[3] = m[3] \* v[0] + m[7] \* v[1] + m[11] \* v[2] + m[15] \* v[3];  
 **return** out;  
 }  
  
  
 **static** *scale*(out, x, y, z) {  
 out[0] \*= x;  
 out[1] \*= x;  
 out[2] \*= x;  
 out[3] \*= x;  
 out[4] \*= y;  
 out[5] \*= y;  
 out[6] \*= y;  
 out[7] \*= y;  
 out[8] \*= z;  
 out[9] \*= z;  
 out[10] \*= z;  
 out[11] \*= z;  
 **return** out;  
 };  
  
 **static** *rotateY*(out, rad) {  
 **let** s = Math.sin(rad),  
 c = Math.cos(rad),  
 a00 = out[0],  
 a01 = out[1],  
 a02 = out[2],  
 a03 = out[3],  
 a20 = out[8],  
 a21 = out[9],  
 a22 = out[10],  
 a23 = out[11];  
  
  
 out[0] = a00 \* c - a20 \* s;  
 out[1] = a01 \* c - a21 \* s;  
 out[2] = a02 \* c - a22 \* s;  
 out[3] = a03 \* c - a23 \* s;  
 out[8] = a00 \* s + a20 \* c;  
 out[9] = a01 \* s + a21 \* c;  
 out[10] = a02 \* s + a22 \* c;  
 out[11] = a03 \* s + a23 \* c;  
 **return** out;  
 }  
  
 **static** *rotateX*(out, rad) {  
 **let** s = Math.sin(rad),  
 c = Math.cos(rad),  
 a10 = out[4],  
 a11 = out[5],  
 a12 = out[6],  
 a13 = out[7],  
 a20 = out[8],  
 a21 = out[9],  
 a22 = out[10],  
 a23 = out[11];  
  
  
 out[4] = a10 \* c + a20 \* s;  
 out[5] = a11 \* c + a21 \* s;  
 out[6] = a12 \* c + a22 \* s;  
 out[7] = a13 \* c + a23 \* s;  
 out[8] = a20 \* c - a10 \* s;  
 out[9] = a21 \* c - a11 \* s;  
 out[10] = a22 \* c - a12 \* s;  
 out[11] = a23 \* c - a13 \* s;  
 **return** out;  
 }  
  
 **static** *rotateZ*(out, rad) {  
 **let** s = Math.sin(rad),  
 c = Math.cos(rad),  
 a00 = out[0],  
 a01 = out[1],  
 a02 = out[2],  
 a03 = out[3],  
 a10 = out[4],  
 a11 = out[5],  
 a12 = out[6],  
 a13 = out[7];  
  
  
 out[0] = a00 \* c + a10 \* s;  
 out[1] = a01 \* c + a11 \* s;  
 out[2] = a02 \* c + a12 \* s;  
 out[3] = a03 \* c + a13 \* s;  
 out[4] = a10 \* c - a00 \* s;  
 out[5] = a11 \* c - a01 \* s;  
 out[6] = a12 \* c - a02 \* s;  
 out[7] = a13 \* c - a03 \* s;  
 **return** out;  
 }  
  
 **static** *invert*(out, mat) {  
 **if** (mat === **undefined**) mat = out;  
  
 **let** a00 = mat[0], a01 = mat[1], a02 = mat[2], a03 = mat[3],  
 a10 = mat[4], a11 = mat[5], a12 = mat[6], a13 = mat[7],  
 a20 = mat[8], a21 = mat[9], a22 = mat[10], a23 = mat[11],  
 a30 = mat[12], a31 = mat[13], a32 = mat[14], a33 = mat[15],  
  
 b00 = a00 \* a11 - a01 \* a10,  
 b01 = a00 \* a12 - a02 \* a10,  
 b02 = a00 \* a13 - a03 \* a10,  
 b03 = a01 \* a12 - a02 \* a11,  
 b04 = a01 \* a13 - a03 \* a11,  
 b05 = a02 \* a13 - a03 \* a12,  
 b06 = a20 \* a31 - a21 \* a30,  
 b07 = a20 \* a32 - a22 \* a30,  
 b08 = a20 \* a33 - a23 \* a30,  
 b09 = a21 \* a32 - a22 \* a31,  
 b10 = a21 \* a33 - a23 \* a31,  
 b11 = a22 \* a33 - a23 \* a32,  
  
  
 det = b00 \* b11 - b01 \* b10 + b02 \* b09 + b03 \* b08 - b04 \* b07 + b05 \* b06;  
  
 **if** (!det) **return false**;  
 det = 1.0 / det;  
  
 out[0] = (a11 \* b11 - a12 \* b10 + a13 \* b09) \* det;  
 out[1] = (a02 \* b10 - a01 \* b11 - a03 \* b09) \* det;  
 out[2] = (a31 \* b05 - a32 \* b04 + a33 \* b03) \* det;  
 out[3] = (a22 \* b04 - a21 \* b05 - a23 \* b03) \* det;  
 out[4] = (a12 \* b08 - a10 \* b11 - a13 \* b07) \* det;  
 out[5] = (a00 \* b11 - a02 \* b08 + a03 \* b07) \* det;  
 out[6] = (a32 \* b02 - a30 \* b05 - a33 \* b01) \* det;  
 out[7] = (a20 \* b05 - a22 \* b02 + a23 \* b01) \* det;  
 out[8] = (a10 \* b10 - a11 \* b08 + a13 \* b06) \* det;  
 out[9] = (a01 \* b08 - a00 \* b10 - a03 \* b06) \* det;  
 out[10] = (a30 \* b04 - a31 \* b02 + a33 \* b00) \* det;  
 out[11] = (a21 \* b02 - a20 \* b04 - a23 \* b00) \* det;  
 out[12] = (a11 \* b07 - a10 \* b09 - a12 \* b06) \* det;  
 out[13] = (a00 \* b09 - a01 \* b07 + a02 \* b06) \* det;  
 out[14] = (a31 \* b01 - a30 \* b03 - a32 \* b00) \* det;  
 out[15] = (a20 \* b03 - a21 \* b01 + a22 \* b00) \* det;  
  
 **return true**;  
 }  
  
 **static** *translate*(out, x, y, z) {  
 out[12] = out[0] \* x + out[4] \* y + out[8] \* z + out[12];  
 out[13] = out[1] \* x + out[5] \* y + out[9] \* z + out[13];  
 out[14] = out[2] \* x + out[6] \* y + out[10] \* z + out[14];  
 out[15] = out[3] \* x + out[7] \* y + out[11] \* z + out[15];  
 }  
}

### Utils

**import** {Colour4f} **from './Colour4f'**;  
**import** {Vec3f} **from './Vec3f'**;  
**import** {Vec2f} **from './Vec2f'**;  
  
*/\*\*  
 \* Вспомогательные методы.  
 \*/***export class** Utils {  
 *// Константы для перевода из градусов в радианы и наоборот* **static** *DEGS\_TO\_RADS* = Math.PI / 180.0;  
 **static** *RADS\_TO\_DEGS* = 180.0 / Math.PI;  
  
 *// Приведение чисел к формату с тремя знаками после запятой* **static** *formatter* = (num) => num.toFixed(3);  
  
 */\*\*  
 \* Некоторые цвета.  
 \*/* **static** *RED* = **new** Colour4f([1.0, 0.0, 0.0, 1.0]);  
 **static** *GREEN* = **new** Colour4f([0.0, 1.0, 0.0, 1.0]);  
 **static** *BLUE* = **new** Colour4f([0.0, 0.0, 1.0, 1.0]);  
 **static** *MID\_RED* = **new** Colour4f([0.6, 0.0, 0.0, 1.0]);  
 **static** *MID\_GREEN* = **new** Colour4f([0.0, 0.6, 0.0, 1.0]);  
 **static** *MID\_BLUE* = **new** Colour4f([0.0, 0.0, 0.6, 1.0]);  
 **static** *BLACK* = **new** Colour4f([0.0, 0.0, 0.0, 1.0]);  
 **static** *GREY* = **new** Colour4f([0.5, 0.5, 0.5, 1.0]);  
 **static** *WHITE* = **new** Colour4f([1.0, 1.0, 1.0, 1.0]);  
 **static** *YELLOW* = **new** Colour4f([1.0, 1.0, 0.0, 1.0]);  
 **static** *CYAN* = **new** Colour4f([0.0, 1.0, 1.0, 1.0]);  
 **static** *MAGENTA* = **new** Colour4f([1.0, 0.0, 1.0, 1.0]);  
  
 */\*\*  
 \* Возвращает котангенс угла в радианах.  
 \* Return the co-tangent of an angle specified in radians.  
 \*  
 \** ***@param*** *angleRads Угол врадианах.  
 \** ***@return*** *Котангенс угла.  
 \*/* **static** *cot* = (angleRads: **number**) => ( 1.0 / Math.tan(angleRads) );  
  
 */\*\*  
 \* Конвертирует из радиан в градусы.  
 \*  
 \** ***@param*** *angleRads Угол в радианах.  
 \** ***@return*** *Угол в градусах.  
 \*/* **static** *radiansToDegrees* = (angleRads: **number**): **number** => angleRads \* Utils.*RADS\_TO\_DEGS*;  
  
 */\*\*  
 \* Конвертирует из градусов в радианы.  
 \*  
 \** ***@param*** *angleDegs Угол в градусах.  
 \** ***@return*** *Угол в радианах.  
 \*/* **static** *degreesToRadians* = (angleDegs: **number**): **number** => angleDegs \* Utils.*DEGS\_TO\_RADS*;  
  
 */\*\*  
 \* Возвращвет знак числа.  
 \*  
 \** ***@param*** *value Число.  
 \** ***@return*** *1 если значение больше или равно нулю, иначе -1.  
 \*/* **static** *sign*(value: **number**): **number** {  
 **if** (value >= 0.0) {  
 **return** 1.0;  
 }  
 **return** -1.0;  
 }  
  
 */\*\*  
 \* Валидирует направление вектора, чтобы убедиться, что он не равен нулю.  
 \* <p>  
 \* Если длина вектора равна нулю бросает Error.  
 \** ***@param*** *directionUV Вектор.  
 \*/* **static** *validateDirectionUV*(directionUV: Vec2f | Vec3f): **void** {  
 **if** ( directionUV.**length**() <= 0.0) {  
 **throw new** Error(**'Vec direction unit vector cannot be zero.'**);  
 }  
 }  
  
 */\*\*  
 \* Валидирует длину кости, чтобы убедаться, что она не нулевая.  
 \* <p>  
 \* Если длина кости равна нулю бросает Error.  
 \** ***@param*** *length Длина.  
 \*/* **static** *validateLength*(length: **number**): **void** {  
 **if** (length < 0.0) {  
 **throw new** Error(**'Length must be a greater than or equal to zero.'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает флаг, показывающий равны ли значения с заданным уровнем терпимости.  
 \*  
 \** ***@param*** *a Первое значение.  
 \** ***@param*** *b Второе значение.  
 \** ***@param*** *tolerance Максимальная разница между числами при которой они будут считаться равными.  
 \** ***@return*** *Равны ли значения.  
 \*/* **static** *approximatelyEquals* = (a: **number**, b: **number**, tolerance: **number**): **boolean** => (Math.*abs*(a - b) <= tolerance);  
}

### Vec2f

**import** {Utils} **from './Utils'**;  
  
*/\*\*  
 \* 2D вектор.  
 \* <p>  
 \* Состоит из x и y полей, задающих направление вектора.  
 \*/***export class** Vec2f {  
 **x**: **number** = 0;  
 **y**: **number** = 0;  
  
 *// ---------- Constructors ----------  
  
 /\*\*  
 \* Конструктор.  
 \* <p>  
 \* Принимает три вариации параметров:  
 \* <p>  
 \* Без параметров. Заполняет x и y значениями по умолчанию (нулями).  
 \* <p>  
 \* Конструктор копирования. Задаёт x и y равными значениям source-вектора.  
 \*  
 \** ***@param*** *source Вектор, параметры которого будут скопированы.  
 \* <p>  
 \* Конструктор с двумя параметрами, задающими x и y.  
 \*  
 \** ***@param*** *source Значение координаты по оси x.  
 \** ***@param*** *y Значение координаты по оси y.  
 \*/* **constructor**(source?: Vec2f | **number**, y?: **number**) {  
 **switch** (***arguments***.**length**) {  
 **case** 0:  
 **return**;  
 **case** 1:  
 **if** (source **instanceof** Vec2f) {  
 **this**.**x** = source.**x**;  
 **this**.**y** = source.**y**;  
 **return**;  
 } **else** {  
 **throw new** Error(**'Wrong Vec2f constructor params!'**);  
 }  
 **case** 2:  
 **if** (**typeof** source === **'number'**) {  
 **this**.**x** = source;  
 **this**.**y** = y;  
 **return**;  
 } **else** {  
 **throw new** Error(**'Wrong Vec2f constructor params!'**);  
 }  
 }  
 }  
  
 *// ---------- Methods ----------  
  
 /\*\*  
 \* Проверяет вектора на примерное равенство.  
 \* Векторы примерно равны если модуль их координат по каждой из осей меньше tolerance.  
 \* <p>  
 \* Если значение tolerance меньше 0 - бросаем ошибку.  
 \*  
 \** ***@param*** *v Вектор для сравнения.  
 \** ***@param*** *tolerance Значение "терпимости" к разнице.  
 \** ***@return*** *Результат сравнения.  
 \*/* approximatelyEquals(v: Vec2f, tolerance: **number**): **boolean** {  
 **if** (tolerance < 0) { **throw new** Error(**'Equality threshold must be greater than or equal to 0.0'**); }  
 **return** (Math.*abs*(**this**.**x** - v.**x**) < tolerance && Math.*abs*(**this**.**y** - v.**y**) < tolerance);  
 }  
  
 */\*\*  
 \* Возвращает сумму данного вектора с вектором из параметров, не меняя данный вектор.  
 \*  
 \** ***@param*** *v Вектор.  
 \** ***@return*** *Результат сложения данного вектора с вектором параметром.  
 \*\*/  
 plus* = (v: Vec2f): Vec2f => **new** Vec2f(**this**.**x** + v.**x**, **this**.**y** + v.**y**);  
  
 */\*\*  
 \* Возвращает разность данного вектора с вектором из параметров, не меняя данный вектор.  
 \*  
 \** ***@param*** *v Вектор.  
 \** ***@return*** *Результат вычитания из данного вектора вектора-параметра.  
 \*\*/  
 minus* = (v: Vec2f): Vec2f => **new** Vec2f(**this**.**x** - v.**x**, **this**.**y** - v.**y**);  
  
 */\*\*  
 \* Умножает вектор на число и возвращает результат, не меняя при этом исходный вектор.  
 \*  
 \** ***@param*** *value Число, на которое необходимо умножить.  
 \** ***@return*** *Результат умножения данного вектора на число.  
 \*\*/  
 times* = (value: **number**): Vec2f => **new** Vec2f(**this**.**x** \* value, **this**.**y** \* value);  
  
 */\*\*  
 \* Делит вектор на число и возвращает результат, не меняя при этом исходный вектор.  
 \*  
 \** ***@param*** *value Число, на которое необходимо поделить.  
 \** ***@return*** *Результат деления данного вектора на число.  
 \*\*/  
 dividedBy* = (value: **number**): Vec2f => **new** Vec2f(**this**.**x** / value, **this**.**y** / value);  
  
 */\*\*  
 \* Возвращает вектор противоположного направления.  
 \* Сам вектор не меняет.  
 \*  
 \** ***@return*** *Вектор противоположного направления.  
 \*/  
 negated* = (): Vec2f => **new** Vec2f(-**this**.**x**, -**this**.**y**);  
  
 */\*\*  
 \* Задаёт значения полей вектора.  
 \* Принимает две вариации параметров:  
 \* <p>  
 \* Один параметр - вектор источник.  
 \*  
 \** ***@param*** *source Вектор-источник.  
 \* <p>  
 \* Два параметра - x и y соответственно.  
 \*  
 \** ***@param*** *source Значение координаты по X.  
 \** ***@param*** *y Значение координаты по Y.  
 \*/* set(source: Vec2f | **number**, y?: **number**): **void** {  
 **switch** (***arguments***.**length**) {  
 **case** 1:  
 **if** (source **instanceof** Vec2f) {  
 **this**.**x** = source.**x**;  
 **this**.**y** = source.**y**;  
 **return**;  
 } **else** {  
 **throw new** Error(**'Wrong Vec2f set params!'**);  
 }  
 **case** 2:  
 **if** (**typeof** source === **'number'**) {  
 **this**.**x** = source;  
 **this**.**y** = y;  
 **return**;  
 } **else** {  
 **throw new** Error(**'Wrong Vec2f set params!'**);  
 }  
 **default**:  
 **throw new** Error(**'Wrong Vec2f set params!'**);  
 }  
 }  
  
 */\*\*  
 \* Возвращает длину вектора.  
 \*  
 \** ***@return*** *Длина вектора.  
 \*/  
 length* = (): **number** => Math.sqrt(**this**.**x** \* **this**.**x** + **this**.**y** \* **this**.**y**);  
  
 */\*\*  
 \* Нормализует и возвращает себя.  
 \*  
 \** ***@return*** *Нормализованный вектор.  
 \*/* normalise(): Vec2f {  
 **const** magnitude = Math.sqrt(**this**.**x** \* **this**.**x** + **this**.**y** \* **this**.**y**);  
  
 **if** (magnitude > 0.0) {  
 **this**.**x** /= magnitude;  
 **this**.**y** /= magnitude;  
 }  
  
 **return this**;  
 }  
  
 */\*\*  
 \* Возвращает нормализованную версию вектора-параметра, не меняясь сам.  
 \*  
 \** ***@param*** *source Вектор для нормализации.  
 \** ***@return*** *Нормализованный вектор.  
 \*/* **static** *normalised*(source: Vec2f): Vec2f { **return new** Vec2f(source).normalise(); }  
  
 */\*\*  
 \* Возвращает направление вектора, для которого a - начальная точка, а b - конечная.  
 \* Направление будет единичным вектором.  
 \*  
 \** ***@param*** *a Начальная точка.  
 \** ***@param*** *b Конечная точка.  
 \** ***@return*** *\*/* **static** *getDirectionUV* = (a: Vec2f, b: Vec2f): Vec2f => b.*minus*(a).normalise();  
  
 */\*\*  
 \* Возвращает расстояние между двумя точками.  
 \*  
 \** ***@param*** *v1 Начальная позиция.  
 \** ***@param*** *v2 Конечная позиция.  
 \** ***@return*** *Расстояние.  
 \*/* **static** *distanceBetween* = (v1: Vec2f, v2:Vec2f): **number** => Math.sqrt( (v2.**x** - v1.**x**) \* (v2.**x** - v1.**x**) + (v2.**y** - v1.**y**) \* (v2.**y** - v1.**y**) );  
  
 */\*\*  
 \* Возвращает скалярное произведение двух векторов параметров.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Скалярное произведение.  
 \*/* **static** *dot* = (v1: Vec2f, v2: Vec2f): **number** => v1.**x** \* v2.**x** + v1.**y** \* v2.**y**;  
  
 */\*\*  
 \* Возвращает скалярное произведение двух векторов: this и вектора-параметра.  
 \*  
 \** ***@param*** *v Второй вектор.  
 \** ***@return*** *Скалярное произведение.  
 \*/  
 dot* = (v: Vec2f): **number** => **this**.**x** \* v.**x** + **this**.**y** \* v.**y**;  
  
 */\*\*  
 \* Возвращает угол между двумя векторами-параметрами.  
 \* У возвращаемого угла нет знака, все значения лежат в проеделах [0..180].  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *float Угол между векторами.  
 \*/* **static** *getUnsignedAngleBetweenVectorsDegs* = (v1: Vec2f, v2: Vec2f): **number** => Math.acos( Vec2f.*normalised*(v1).*dot*( Vec2f.*normalised*(v2) ) ) \* Utils.*RADS\_TO\_DEGS*;  
  
 */\*\*  
 \* Возвращает знак угла между двумя векторами.  
 \* 1, если второй вектор находится против часовой стрелки относительно первого;  
 \* -1, если второй вектор находится по часовой стрелке относительно первого;  
 \* 0, если угол между ними 0.  
 \*  
 \** ***@param*** *u Первый угол.  
 \** ***@param*** *v Второй угол.  
 \** ***@return*** *Знак угла.  
 \*/* **static** *zcross*(u: Vec2f, v: Vec2f): **number** {  
 **const** p = u.**x** \* v.**y** - v.**x** \* u.**y**;  
  
 **if** (p > 0.0) {  
 **return** 1;  
 } **else** {  
 **if** (p < 0.0) {  
 **return** -1;  
 }  
 }  
  
 **return** 0;  
 }  
  
 */\*\*  
 \* Возвращает угол со знаком между двумя векорами: this и параметром.  
 \* <p>  
 \* Угол будет положительным, если второй вектор находится против часовой стрелки относительно первого;  
 \* Угол будет отричательным, если второй вектор находится по часовой стрелке относительно первого.  
 \*  
 \** ***@param*** *otherVector Второй вектор.  
 \** ***@return*** *float Угол между векторами в градусах.  
 \*/* **public** getSignedAngleDegsTo(otherVector: Vec2f): **number** {  
 **const** thisVectorUV = Vec2f.*normalised*(**this**);  
 **const** otherVectorUV = Vec2f.*normalised*(otherVector);  
  
 **const** unsignedAngleDegs = Math.acos( thisVectorUV.*dot*(otherVectorUV) ) \* Utils.*RADS\_TO\_DEGS*;  
  
 **if** ( Vec2f.*zcross*(thisVectorUV, otherVectorUV) == 1) {  
 **return** unsignedAngleDegs;  
 } **else** {  
 **return** -unsignedAngleDegs;  
 }  
 }  
  
 */\*\*  
 \* Возвращает единичный вектор с примененными к нему ограничениями.  
 \* Если угол между идеальным положением вектора и бейзлайном привышает ограничения, то  
 \* возвращаем вектор, который повернут на граничный угол относительно бейзлайна.  
 \* В противном случае - возвращаем идеальный вектор.  
 \*  
 \** ***@param*** *directionUV Идеальное положение вектора, которое возможно недостижимо.  
 \** ***@param*** *baselineUV Вектор относительно которого считаются углы.  
 \** ***@param*** *clockwiseConstraintDegs Угол-ограничение по часовой стрелке.  
 \** ***@param*** *antiClockwiseConstraintDegs Угол-ограничение против часовой стрелки.  
 \** ***@return*** *Vec2f  
 \*/* **static** *getConstrainedUV*(directionUV: Vec2f, baselineUV: Vec2f, clockwiseConstraintDegs: **number**, antiClockwiseConstraintDegs: **number**): Vec2f {  
 **const** signedAngleDegs = baselineUV.getSignedAngleDegsTo(directionUV);  
  
 **if** (signedAngleDegs > antiClockwiseConstraintDegs) {  
 **return** Vec2f.*rotateDegs*(baselineUV, antiClockwiseConstraintDegs);  
 }  
  
 **if** (signedAngleDegs < -clockwiseConstraintDegs) {  
 **return** Vec2f.*rotateDegs*(baselineUV, -clockwiseConstraintDegs);  
 }  
  
 **return** directionUV;  
 }  
  
 */\*\*  
 \* Вращает текущий вектор на число-параметр радиан.  
 \* <p>  
 \* Положительное значение угла свидетельствует о повороте против часовой стрелки.  
 \* Отрицательное значеине - по часовой.  
 \* <p>  
 \* Меняет текущий вектор и возвращает this для создания цепочек.  
 \** ***@param*** *angleRads TУгол поворота в радианах.  
 \** ***@return*** *Vec2f  
 \*/* rotateRads(angleRads: **number**): Vec2f {  
 *// Поворот вокруг оси z:  
 // x' = x\*cos q - y\*sin q  
 // y' = x\*sin q + y\*cos q  
 // z' = z* **const** cosTheta = Math.cos(angleRads);  
 **const** sinTheta = Math.sin(angleRads);  
  
 **const** rotatedVector = **new** Vec2f(**this**.**x** \* cosTheta - **this**.**y** \* sinTheta, *// x* **this**.**x** \* sinTheta + **this**.**y** \* cosTheta); *// y* **this**.**x** = rotatedVector.**x**;  
 **this**.**y** = rotatedVector.**y**;  
  
 **return this**;  
 }  
  
 */\*\*  
 \* Поворачивает вектор-параметр на заданное число градусов.  
 \* Method to rotate this Vec2f by a given angle as specified in degrees.  
 \* <p>  
 \* Метод не меняет исходный вектор.  
 \* Положительное значение угла свидетельствует о повороте против часовой стрелки.  
 \* Отрицательное значеине - по часовой.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол поворота в градусах.  
 \** ***@return*** *Vec2f  
 \*/* **static** *rotateDegs*(source: Vec2f, angleDegs: **number**): Vec2f {  
 *// Поворот вокруг оси z:  
 // x' = x\*cos q - y\*sin q  
 // y' = x\*sin q + y\*cos q  
 // z' = z* **const** angleRads = angleDegs \* Utils.*DEGS\_TO\_RADS*;  
  
 **const** cosTheta = Math.cos(angleRads);  
 **const** sinTheta = Math.sin(angleRads);  
  
 **return new** Vec2f(source.**x** \* cosTheta - source.**y** \* sinTheta, *// x* source.**x** \* sinTheta + source.**y** \* cosTheta); *// y* }  
  
 *toArray* = (): **number**[] => [**this**.**x**, **this**.**y**];  
}

### Vec3f

**import** {Utils} **from './Utils'**;  
**import** {Mat3f} **from './Mat3f'**;  
  
*/\*\*  
 \* 3D вектор.  
 \*/***export class** Vec3f {  
 *// Координатные оси* **static** *X\_AXIS* = **new** Vec3f(1, 0, 0);  
 **static** *Y\_AXIS* = **new** Vec3f(0, 1, 0);  
 **static** *Z\_AXIS* = **new** Vec3f(0, 0, 1);  
  
 *// ----- Properties -----  
  
 // Координаты вектора* **x**: **number** = 0;  
 **y**: **number** = 0;  
 **z**: **number** = 0;  
  
 *// ----- Methods -----  
  
 /\*\*  
 \* Конструктор, который принимает четыре вариации параметров:  
 \* <p>  
 \* Без параметров - все поля принимают значения по умолчанию.  
 \* <p>  
 \* Конструктор копирования. Создается глубокая копия объекта.  
 \*  
 \** ***@param*** *value Вектор-источник  
 \* <p>  
 \* Конструктор с одним параметром - числом, задающим координаты вектора по всем осям.  
 \*  
 \** ***@param*** *value Координата вектора по каждой из осей.  
 \* <p>  
 \* Конструктор с тремя параметрами - значения координат по X, Y, Z осям соответственно.  
 \*  
 \** ***@param*** *value Значении координаты по оси X.  
 \** ***@param*** *y Значении координаты по оси Y.  
 \** ***@param*** *z Значении координаты по оси Z.  
 \* \*/* **constructor**(value?: **number** | Vec3f, y?: **number**, z?: **number**) {  
 **switch** (***arguments***.**length**) {  
 **case** 0: **break**;  
 **case** 1:  
 **if** (value **instanceof** Vec3f) {  
 **this**.**x** = value.**x**;  
 **this**.**y** = value.**y**;  
 **this**.**z** = value.**z**;  
 } **else** {  
 **this**.**x** = **this**.**y** = **this**.**z** = value;  
 }  
 **break**;  
 **case** 3:  
 **if** (**typeof** value === **'number'**) {  
 **this**.**x** = value;  
 **this**.**y** = y;  
 **this**.**z** = z;  
 } **else** {  
 **throw new** Error(**'Vec3f constructor was invoked incorrectly.'**)  
 }  
 **break**;  
 }  
 }  
  
 */\*\*  
 \* Возвращает копию вектора-источника.  
 \*  
 \** ***@param*** *source Вектор-источник.  
 \** ***@return*** *Глубокая копия вектора источника .  
 \*/* **static** *clone* = (source: Vec3f): Vec3f => **new** Vec3f(source.**x**, source.**y**, source.**z**);  
  
 */\*\*  
 \* Задаёт значения полей вектора.  
 \* Принимает две вариации параметров:  
 \* <p>  
 \* Один параметр - вектор источник.  
 \*  
 \** ***@param*** *source Вектор-источник.  
 \* <p>  
 \* Три параметра - x, y, z соответственно.  
 \*  
 \** ***@param*** *source Значение координаты по X.  
 \** ***@param*** *y Значение координаты по Y.  
 \** ***@param*** *z Значение координаты по Z.  
 \*/* set(source: Vec3f | **number**, y?: **number**, z?: **number**): **void** {  
 **switch** (***arguments***.**length**) {  
 **case** 0: **break**;  
 **case** 1:  
 **if** (source **instanceof** Vec3f) {  
 **this**.**x** = source.**x**;  
 **this**.**y** = source.**y**;  
 **this**.**z** = source.**z**;  
 } **else** {  
 **throw new** Error(**'Vec3f set was invoked incorrectly.'**)  
 }  
 **break**;  
 **case** 3:  
 **if** (**typeof** source === **'number'**) {  
 **this**.**x** = source;  
 **this**.**y** = y;  
 **this**.**z** = z;  
 } **else** {  
 **throw new** Error(**'Vec3f set was invoked incorrectly.'**)  
 }  
 **break**;  
 **default**:  
 **throw new** Error(**'Vec3f set was invoked incorrectly.'**)  
 }  
 }  
  
 */\*\*  
 \* Проверяет вектора на примерное равенство.  
 \* Векторы примерно равны если модуль их координат по каждой из осей меньше tolerance.  
 \* <p>  
 \* Если значение tolerance меньше 0 - бросаем ошибку.  
 \*  
 \** ***@param*** *v Вектор для сравнения.  
 \** ***@param*** *tolerance Значение "терпимости" к разнице.  
 \** ***@return*** *Результат сравнения.  
 \*/* approximatelyEquals(v, tolerance) {  
 **if** (tolerance < 0) {  
 **throw** Error(**'Equality threshold must be greater than or equal to 0.0f'**);  
 }  
  
 **const** xDiff = Math.*abs*(**this**.**x** - v.**x**);  
 **const** yDiff = Math.*abs*(**this**.**y** - v.**y**);  
 **const** zDiff = Math.*abs*(**this**.**z** - v.**z**);  
  
 **return** (xDiff < tolerance && yDiff < tolerance && zDiff < tolerance);  
 }  
  
 */\*\*  
 \* Возвращает перпендикулярны ли два вектора.  
 \*  
 \** ***@param*** *a Первый вектор.  
 \** ***@param*** *b Второй вектор.  
 \** ***@return*** *Перпендикулярны ли они.  
 \*/* **static** *perpendicular* = (a: Vec3f, b: Vec3f): **boolean** => Utils.*approximatelyEquals*( Vec3f.*dotProduct*(a, b), 0.0, 0.01 );  
  
 */\*\*  
 \* Возвращает равна ли примерно длина данного вектора числу параметру.  
 \*  
 \** ***@param*** *value Значение для сравнения с длиной.  
 \** ***@param*** *tolerance Значение "терпимости".  
 \** ***@return*** *Равны ли примерно длина и число-параметр.  
 \*/* lengthIsApproximately(value: **number**, tolerance: **number**): **boolean** {  
 **if** (tolerance < 0) {  
 **throw new** Error(**'Comparison tolerance cannot be less than zero.'**);  
 }  
  
 **return** (Math.*abs*(**this**.*length*() - value) < tolerance);  
 }  
  
 */\*\* Зануляет координаты данного вектора \*/* zero(): **void** { **this**.**x** = **this**.**y** = **this**.**z** = 0.0; }  
  
 */\*\*  
 \* Задаёт текущему вектору противоположное направление. Возвращает this для построения цепочек.  
 \*  
 \** ***@return*** *Вектор с противоположным направлением.  
 \*/* negate(): Vec3f {  
 **this**.**x** = -**this**.**x**;  
 **this**.**y** = -**this**.**y**;  
 **this**.**z** = -**this**.**z**;  
  
 **return this**;  
 }  
  
 */\*\*  
 \* Возвращает вектор противоположного направления.  
 \* Сам вектор не меняет.  
 \*  
 \** ***@return*** *Вектор противоположного направления.  
 \*/  
 negated* = (): Vec3f => **new** Vec3f(-**this**.**x**, -**this**.**y**, -**this**.**z**);  
  
 */\*\*  
 \* Возвращает равны ли примерно вектора при заданном уровне "терпимости".  
 \*  
 \** ***@param*** *a Первый вектор.  
 \** ***@param*** *b Второй вектор.  
 \** ***@param*** *tolerance Уровень "терпимости", максимальная разница между векторами, при которой они всё еще равны.  
 \** ***@return*** *\*/* **static** *approximatelyEqual* = (a: Vec3f, b: Vec3f, tolerance: **number**): **boolean** => (  
 (Math.*abs*(a.**x** - b.**x**) < tolerance)  
 && (Math.*abs*(a.**y** - b.**y**) < tolerance)  
 && (Math.*abs*(a.**z** - b.**z**) < tolerance)  
 );  
  
 */\*\*  
 \* Нормализует и возвращает себя.  
 \* Если величина вектора равна нулю, то возвращается вектор без изменений.  
 \*  
 \** ***@return*** *Нормализованный вектор.  
 \*/* normalise(): Vec3f {  
 **const** magnitude = Math.sqrt(**this**.**x** \* **this**.**x** + **this**.**y** \* **this**.**y** + **this**.**z** \* **this**.**z**);  
  
 **if** (magnitude > 0) {  
 **this**.**x** /= magnitude;  
 **this**.**y** /= magnitude;  
 **this**.**z** /= magnitude;  
 }  
  
 **return this**;  
 }  
  
 */\*\*  
 \* Возвращает нормализованную версию данного вектора, не меняясь сам.  
 \*  
 \** ***@return*** *Нормализованный вектор.  
 \*/* normalised(): Vec3f { **return new** Vec3f(**this**).normalise(); }  
  
 */\*\*  
 \* Возвращает скалярное произведение двух векторов переданных в параметрах.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Скалярное произведение.  
 \*/* **static** *scalarProduct* = (v1: Vec3f, v2: Vec3f): **number** => v1.**x** \* v2.**x** + v1.**y** \* v2.**y** + v1.**z** \* v2.**z**;  
  
 */\*\*  
 \* Возвращает скалярное произведение двух нормализованных векторов переданных в параметрах.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Скалярное произведение нормализованных векторов.  
 \*/* **static** *dotProduct*(v1: Vec3f, v2: Vec3f): **number** {  
 **const** v1Norm = v1.normalised();  
 **const** v2Norm = v2.normalised();  
  
 **return** v1Norm.**x** \* v2Norm.**x** + v1Norm.**y** \* v2Norm.**y** + v1Norm.**z** \* v2Norm.**z**;  
 }  
  
 */\*\*  
 \* Возвращает векторное произведение двух векторов переданных в параметрах.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Векторное произведение векторов.  
 \*/* **static** *crossProduct* = (v1: Vec3f, v2: Vec3f): Vec3f => **new** Vec3f(v1.**y** \* v2.**z** - v1.**z** \* v2.**y**, v1.**z** \* v2.**x** - v1.**x** \* v2.**z**, v1.**x** \* v2.**y** - v1.**y** \* v2.**x**);  
  
 */\*\*  
 \* Возвращает векторное произведение двух векторов: this и параметра.  
 \*  
 \** ***@param*** *v Второй вектор.  
 \** ***@return*** *Скалярное произведение нормализованных векторов.  
 \*/  
 cross* = (v: Vec3f): Vec3f => **new** Vec3f(**this**.**y** \* v.**z** - **this**.**z** \* v.**y**, **this**.**z** \* v.**x** - **this**.**x** \* v.**z**, **this**.**x** \* v.**y** - **this**.**y** \* v.**x**);  
  
 */\*\*  
 \* Вычисляет расстояние между двумя точками в 3D.  
 \*  
 \** ***@param*** *v1 Первая точка.  
 \** ***@param*** *v2 Вторая точка.  
 \** ***@return*** *Расстояние между точками.  
 \*/* **static** *distanceBetween*(v1: Vec3f, v2: Vec3f) : **number** {  
 **const** dx = v2.**x** - v1.**x**;  
 **const** dy = v2.**y** - v1.**y**;  
 **const** dz = v2.**z** - v1.**z**;  
 **return** Math.sqrt(dx \* dx + dy \* dy + dz \* dz);  
 }  
  
 */\*\*  
 \* Возвращает длину вектора.  
 \*  
 \** ***@return*** *Длина вектора.  
 \*/  
 length* = (): **number** => Math.sqrt(**this**.**x** \* **this**.**x** + **this**.**y** \* **this**.**y** + **this**.**z** \* **this**.**z**);  
  
 */\*\*  
 \* Возвращает вектор, компоненты которого - это абсолютные значения вектора-параметра.  
 \* Не меняет исходный вектор.  
 \*  
 \** ***@param*** *source Вектор.  
 \** ***@return*** *Покомпонентно абсолютный аналог вектора параметра.  
 \*/* **static** *abs*(source: Vec3f): Vec3f {  
 **const** absVector = **new** Vec3f();  
  
 **if** (source.**x** < 0.0) { absVector.**x** = -source.**x**; } **else** { absVector.**x** = source.**x**; }  
 **if** (source.**y** < 0.0) { absVector.**y** = -source.**y**; } **else** { absVector.**y** = source.**y**; }  
 **if** (source.**z** < 0.0) { absVector.**z** = -source.**z**; } **else** { absVector.**z** = source.**z**; }  
  
 **return** absVector;  
 }  
  
 */\*\*  
 \* Возвращает единичный вектор-перпендикуляр к вектору параметру.  
 \*  
 \** ***@param*** *u Вектор.  
 \** ***@return*** *Единичный вектор-перпендикуляр для вектора-параметра.  
 \*/* **static** *genPerpendicularVectorQuick*(u: Vec3f): Vec3f {  
 **let** perp: Vec3f;  
  
 **if** (Math.*abs*(u.**y**) < 0.99) {  
 perp = **new** Vec3f(-u.**z**, 0.0, u.**x**); *// cross(u, UP)* }  
 **else** {  
 perp = **new** Vec3f(0.0, u.**z**, -u.**y**); *// cross(u, RIGHT)* }  
  
 **return** perp.normalise();  
 }  
  
 */\*\*  
 \* Возвращает единичный вектор-направление от первой точки ко второй.  
 \*  
 \** ***@param*** *v1 Первая точка.  
 \** ***@param*** *v2 Вторая точка.  
 \** ***@return*** *Единичный вектор-направление.  
 \*/* **static** *getUvBetween* = (v1: Vec3f, v2: Vec3f): Vec3f => **new** Vec3f( v2.*minus*(v1) ).normalise();  
  
 */\*\*  
 \* Возвращает вектор в радианах между векторами.  
 \* <p>  
 \* Результат будет всегда положительным числом от 0 до pi (3.14159) радиан.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Угол между векторами в радианах.  
 \*/* **static** *getAngleBetweenRads*(v1: Vec3f, v2: Vec3f): **number** {  
 *// Note: a and b are normalised within the dotProduct method.* **return** Math.acos( Vec3f.*dotProduct*(v1, v2) );  
 }  
  
 */\*\*  
 \* Возвращает вектор в градусах между векторами.  
 \* <p>  
 \* Результат будет всегда положительным числом от 0 до 180.  
 \*  
 \** ***@param*** *v1 Первый вектор.  
 \** ***@param*** *v2 Второй вектор.  
 \** ***@return*** *Угол между векторами в градусах.  
 \*/* **static** *getAngleBetweenDegs*(v1: Vec3f, v2: Vec3f): **number** { **return** Vec3f.*getAngleBetweenRads*(v1, v2) \* Utils.*RADS\_TO\_DEGS*; }  
  
 */\*\*  
 \* Возвращает угол между векторами [-179.9..180.0].  
 \*  
 \** ***@param*** *referenceVector Первый, базовый вектор, от которого ведется отсчет.  
 \** ***@param*** *otherVector Второй вектор.  
 \** ***@param*** *normalVector Вектор, первендикулярный первым двум.  
 \** ***@return*** *Угол между векторами.  
 \*\*/* **static** *getSignedAngleBetweenDegs*(referenceVector: Vec3f, otherVector: Vec3f, normalVector: Vec3f): **number** {  
 **const** unsignedAngle = Vec3f.*getAngleBetweenDegs*(referenceVector, otherVector);  
 **const** sign = Utils.*sign*( Vec3f.*dotProduct*(Vec3f.*crossProduct*(referenceVector, otherVector), normalVector));  
 **return** unsignedAngle \* sign;  
 }  
  
 */\*\*  
 \* Возвращает вектор с применёнными к нему ограничениями.  
 \*  
 \** ***@param*** *vecToLimit Вектор поворот которого надо ограничить по отношению к базовому вектору.  
 \** ***@param*** *vecBaseline Базовый вектор.  
 \** ***@param*** *angleLimitDegs Максимально допустимый угол в градусах.  
 \** ***@return*** *Ограниченный вектор.  
 \*/* **static** *getAngleLimitedUnitVectorDegs*(vecToLimit: Vec3f, vecBaseline: Vec3f, angleLimitDegs: **number**): Vec3f {  
 **const** angleBetweenVectorsDegs = Vec3f.*getAngleBetweenDegs*(vecBaseline, vecToLimit);  
  
 **if** (angleBetweenVectorsDegs > angleLimitDegs) {  
 **const** correctionAxis = Vec3f.*crossProduct*(vecBaseline.normalised(), vecToLimit.normalised() ).normalise();  
  
 **return** Vec3f.*rotateAboutAxisDegs*(vecBaseline, angleLimitDegs, correctionAxis).normalised();  
 } **else** {  
 **return** vecToLimit.normalised();  
 }  
 }  
  
 */\*\*  
 \* Возвращает глобальный поворот этого вектора относительно глобальной оси X.  
 \* Возвращаемое значение находится в диапазоне -179.9..180.0.  
 \*  
 \** ***@return*** *Отклонение вектора в градусах.  
 \*\*/* getGlobalPitchDegs(): **number** {  
 **const** xProjected = **this**.projectOntoPlane(Vec3f.*X\_AXIS*);  
 **const** pitch = Vec3f.*getAngleBetweenDegs*( Vec3f.*Z\_AXIS*.*negated*(), xProjected);  
 **return** xProjected.**y** < 0.0 ? -pitch : pitch;  
 }  
  
 */\*\*  
 \* Возвращает глобальный поворот этого вектора относительно глобальной оси Y.  
 \* Возвращаемое значение находится в диапазоне -179.9..180.0.  
 \*  
 \** ***@return*** *Отклонение вектора в градусах.  
 \*\*/* getGlobalYawDegs(): **number** {  
 **const** yProjected = **this**.projectOntoPlane(Vec3f.*Y\_AXIS*);  
 **const** yaw = Vec3f.*getAngleBetweenDegs*( Vec3f.*Z\_AXIS*.*negated*(), yProjected);  
 **return** yProjected.**x** < 0.0 ? -yaw : yaw;  
 }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси X на заданный угол в радианах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleRads Угол в радианах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateXRads*(source: Vec3f, angleRads: **number**): Vec3f {  
 *// Поворот вокруг оси x:  
 // x' = x  
 // y' = y\*cos q - z\*sin q  
 // z' = y\*sin q + z\*cos q* **const** cosTheta = Math.cos(angleRads);  
 **const** sinTheta = Math.sin(angleRads);  
  
 **return new** Vec3f(source.**x**, source.**y** \* cosTheta - source.**z** \* sinTheta, source.**y** \* sinTheta + source.**z** \* cosTheta);  
 }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси X на заданный угол в градусах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол в градусах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateXDegs*(source: Vec3f, angleDegs: **number**): Vec3f { **return** Vec3f.*rotateXRads*(source, angleDegs \* Utils.*DEGS\_TO\_RADS*); }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси Y на заданный угол в радианах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleRads Угол в радианах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateYRads*(source: Vec3f, angleRads: **number**): Vec3f {  
 *// Поворот вокруг оси y:  
 // x' = z\*sin q + x\*cos q  
 // y' = y  
 // z' = z\*cos q - x\*sin q* **const** cosTheta = Math.cos(angleRads);  
 **const** sinTheta = Math.sin(angleRads);  
   
 **return new** Vec3f(source.**z** \* sinTheta + source.**x** \* cosTheta, source.**y**, source.**z** \* cosTheta - source.**x** \* sinTheta);  
 }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси Y на заданный угол в градусах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол в градусах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateYDegs*(source: Vec3f, angleDegs: **number**): Vec3f { **return** Vec3f.*rotateYRads*(source, angleDegs \* Utils.*DEGS\_TO\_RADS*); }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси Z на заданный угол в радианах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleRads Угол в радианах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateZRads*(source: Vec3f, angleRads: **number**): Vec3f {  
 *// Поворот вокруг оси z:  
 // x' = x\*cos q - y\*sin q  
 // y' = x\*sin q + y\*cos q  
 // z' = z* **const** cosTheta = Math.cos(angleRads);  
 **const** sinTheta = Math.sin(angleRads);  
  
 **return new** Vec3f(source.**x** \* cosTheta - source.**y** \* sinTheta, source.**x** \* sinTheta + source.**y** \* cosTheta, source.**z**);  
 }  
  
 */\*\*  
 \* Поворачивает вектор-параметр вокруг оси Z на заданный угол в градусах.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол в градусах.  
 \** ***@return*** *Повернутый вектор.  
 \*/* **static** *rotateZDegs*(source: Vec3f, angleDegs: **number**): Vec3f { **return** Vec3f.*rotateZRads*(source, angleDegs \* Utils.*DEGS\_TO\_RADS*); }  
  
 */\*\*  
 \* Поворачивает вектор относительно оси.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол поворота в радианах.  
 \** ***@param*** *rotationAxis Ось вращения.  
 \** ***@return*** *Исходный вектор, повернутый на нужный угол.  
 \*/* **static** *rotateAboutAxisRads*(source: Vec3f, angleRads: **number**, rotationAxis: Vec3f): Vec3f {  
 **const** rotationMatrix = **new** Mat3f();  
  
 **const** sinTheta = Math.sin(angleRads);  
 **const** cosTheta = Math.cos(angleRads);  
 **const** oneMinusCosTheta = 1.0 - cosTheta;  
  
 **const** xyOne = rotationAxis.**x** \* rotationAxis.**y** \* oneMinusCosTheta;  
 **const** xzOne = rotationAxis.**x** \* rotationAxis.**z** \* oneMinusCosTheta;  
 **const** yzOne = rotationAxis.**y** \* rotationAxis.**z** \* oneMinusCosTheta;  
  
 rotationMatrix.**m00** = rotationAxis.**x** \* rotationAxis.**x** \* oneMinusCosTheta + cosTheta;  
 rotationMatrix.**m01** = xyOne + rotationAxis.**z** \* sinTheta;  
 rotationMatrix.**m02** = xzOne - rotationAxis.**y** \* sinTheta;  
  
 rotationMatrix.**m10** = xyOne - rotationAxis.**z** \* sinTheta;  
 rotationMatrix.**m11** = rotationAxis.**y** \* rotationAxis.**y** \* oneMinusCosTheta + cosTheta;  
 rotationMatrix.**m12** = yzOne + rotationAxis.**x** \* sinTheta;  
  
 rotationMatrix.**m20** = xzOne + rotationAxis.**y** \* sinTheta;  
 rotationMatrix.**m21** = yzOne - rotationAxis.**x** \* sinTheta;  
 rotationMatrix.**m22** = rotationAxis.**z** \* rotationAxis.**z** \* oneMinusCosTheta + cosTheta;  
  
 **return** rotationMatrix.times(source);  
 }  
  
 */\*\*  
 \* Поворачивает вектор относительно оси.  
 \*  
 \** ***@param*** *source Вектор для поворота.  
 \** ***@param*** *angleDegs Угол поворота в градусах.  
 \** ***@param*** *rotationAxis Ось вращения.  
 \** ***@return*** *Исходный вектор, повернутый на нужный градус.  
 \*/* **static** *rotateAboutAxisDegs*(source: Vec3f, angleDegs: **number**, rotationAxis: Vec3f): Vec3f {  
 **return** Vec3f.*rotateAboutAxisRads*(source, angleDegs \* Utils.*DEGS\_TO\_RADS*, rotationAxis);  
 }  
  
 */\*\*  
 \* Возвращает сумму данного вектора с вектором из параметров, не меняя данный вектор.  
 \*  
 \** ***@param*** *v Вектор.  
 \** ***@return*** *Результат сложения данного вектора с вектором параметром.  
 \*\*/  
 plus* = (v: Vec3f): Vec3f => **new** Vec3f(**this**.**x** + v.**x**, **this**.**y** + v.**y**, **this**.**z** + v.**z**);  
  
 */\*\*  
 \* Возвращает разность данного вектора с вектором из параметров, не меняя данный вектор.  
 \*  
 \** ***@param*** *v Вектор.  
 \** ***@return*** *Результат вычитания из данного вектора вектора-параметра.  
 \*\*/  
 minus* = (v: Vec3f): Vec3f => **new** Vec3f(**this**.**x** - v.**x**, **this**.**y** - v.**y**, **this**.**z** - v.**z**);  
  
 */\*\*  
 \* Возвращает новый вектор, у которого каждая компонента домноженный на число или  
 \* на соответствующую компоненту вектора-параметра.  
 \*  
 \** ***@param*** *scale Число или вектор.  
 \** ***@return*** *\*\*/* times(scale: Vec3f | **number**): Vec3f {  
 **if** (scale **instanceof** Vec3f) {  
 **return new** Vec3f(**this**.**x** \* scale.**x**, **this**.**y** \* scale.**y**, **this**.**z** \* scale.**z**);  
 } **else** {  
 **return new** Vec3f(**this**.**x** \* scale, **this**.**y** \* scale, **this**.**z** \* scale);  
 }  
 }  
  
 */\*\*  
 \* Домножает каждую компоненту вектора-параметра на число-параметр.  
 \*  
 \** ***@param*** *v Вектор.  
 \** ***@param*** *scale Число, на которое надо домножить.  
 \*\*/* **static** *times*(v: Vec3f, scale: **number**): **void** { v.**x** \*= scale; v.**y** \*= scale; v.**z** \*= scale; }  
  
 */\*\*  
 \* Покомпонентно складывает два вектора, записывая результат в первый.  
 \*  
 \** ***@param*** *source Вектор, в который запишется результат сложения.  
 \** ***@param*** *other Второй вектор.  
 \*/* **static** *add*(source: Vec3f, other: Vec3f): **void** { source.**x** += other.**x**; source.**y** += other.**y**; source.**z** += other.**z**; }  
  
 */\*\*  
 \* Покомпонентно вычитает из первого вектора второй, модифицируя первый.  
 \*  
 \** ***@param*** *source Вектор, который будет содержать результат.  
 \** ***@param*** *other Второй вектор.  
 \*/* **static** *subtract*(source: Vec3f, other: Vec3f): **void** { source.**x** -= other.**x**; source.**y** -= other.**y**; source.**z** -= other.**z**; }  
  
 */\*\*  
 \* Делит вектор на число и возвращает результат, не меняя при этом исходный вектор.  
 \*  
 \** ***@param*** *value Число, на которое необходимо поделить.  
 \** ***@return*** *Результат деления данного вектора на число.  
 \*\*/* dividedBy(value: **number**): Vec3f { **return new** Vec3f(**this**.**x** / value, **this**.**y** / value, **this**.**z** / value); }  
  
 */\*\*  
 \* Возвращает проекцию вектора на плоскость.  
 \*  
 \** ***@param*** *planeNormal Нормаль к плоскости.  
 \** ***@return*** *Проекция вектора.  
 \*/* projectOntoPlane(planeNormal: Vec3f): Vec3f {  
 **if** ( !(planeNormal.*length*() > 0.0) ) { **throw new** Error(**'Plane normal cannot be a zero vector.'**); }  
  
 *// Проекция вектора b на плоскость с нормалью n : b - ( b.n / ( |n| squared )) \* n* **const** b = **this**.normalised();  
 **const** n = planeNormal.normalised();  
 **return** b.*minus*(n.times(Vec3f.*dotProduct*(b, planeNormal))).normalise();  
 }  
  
 */\*\*  
 \* Возвращает направление вектора, для которого v1 - начальная точка, а v2 - конечная.  
 \* Направление будет единичным вектором.  
 \*  
 \** ***@param*** *v1 Начальная точка.  
 \** ***@param*** *v2 Конечная точка.  
 \** ***@return*** *\*/* **static** *getDirectionUV*(v1: Vec3f, v2: Vec3f): Vec3f { **return** v2.*minus*(v1).normalise(); }  
  
 *toArray* = (): **number**[] => [**this**.**x**, **this**.**y**, **this**.**z**];  
}

# ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ

1. ГОСТ 19.103-77 Обозначения программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
2. ГОСТ 19.104-78 Основные надписи. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
3. ГОСТ 19.105-78 Общие требования к программным документам. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
4. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
5. ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.603-78 Общие правила внесения изменений. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.604-78 Правила внесения изменений в программные документы, выполненные печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

# 

# ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номера листов (страниц) | | | | | Всего листов (страниц в докум.) | № документа | Входящий № сопроводительного докум. и дата | Подп. | Дата |
| Изм. | Измененных | Замененных | Новых | Аннулированных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |