GTS - Ginren Traffic System

銀河連邦

2025/06/04

目次

[1. 交通信号制御機 1](#_Toc199942822)

[1-1. パックJSON（コンフィグ）の仕様 1](#_Toc199942823)

[1-2. 制御方式と用語 1](#_Toc199942824)

[1-2-1. サイクル 1](#_Toc199942825)

[1-2-2. フェーズ 1](#_Toc199942826)

[1-2-3. チャンネル 1](#_Toc199942827)

[1-3. 制御情報のJSONファイルについて（中級者向け） 1](#_Toc199942828)

[1-4. カスタム制御スクリプトについて（上級者向け） 1](#_Toc199942829)

[1-4-1. カスタムスクリプトの仕様 1](#_Toc199942830)

[1-4-2. カスタムスクリプトの例 1](#_Toc199942831)

[1-4-3. カスタムスクリプトの指定方法 1](#_Toc199942832)

[2. 交通信号機 1](#_Toc199942833)

[2-1. パックJSON（コンフィグ）の仕様 1](#_Toc199942834)

[2-1-1. textures 1](#_Toc199942835)

[2-1-2. patterns 1](#_Toc199942836)

[2-1-3. 例 1](#_Toc199942837)

[2-2. チャンネルについて 1](#_Toc199942838)

[3. 交通信号柱 1](#_Toc199942839)

[3-1. パックJSON（コンフィグ）の仕様 1](#_Toc199942840)

[3-1-1. 例 1](#_Toc199942841)

[3-2. モデルの作り方 1](#_Toc199942842)

[4. アーム 1](#_Toc199942843)

[4-1. パックJSON（コンフィグ）の仕様 1](#_Toc199942844)

[4-1-1. 例 1](#_Toc199942845)

[4-2. モデルの作り方 1](#_Toc199942846)

[4-2-1. 描画の仕組み 1](#_Toc199942847)

[4-2-2. baseObjectsの制約 1](#_Toc199942848)

[4-2-3. その他の制約 1](#_Toc199942849)

# 交通信号制御機

最終更新日時：2025年6月4日

交通信号制御機（以下、制御機と記します）は、以下のクラスにより構成されています。

* GTSBlockTrafficController → ブロックインスタンス
* GTSTileEntityTrafficController → 制御機のTileEntity
* GTSTileEntityRendererTrafficController → 制御機のレンダラー
* GTSTrafficControllerConfig → 制御機の個別情報（パック）

Version 2.0-alpha2の時点では、制御機のモデル変更には対応していません。

## パックJSON（コンフィグ）の仕様

現時点で、制御機は共通の設定項目以外の独自設定項目は存在しません。

## 制御方式と用語

制御機は、サイクル フェーズ チャンネル の3つの要素を用いて、信号機の制御を模倣します。この概念について、以下のような「押ボタン式横断路」の動作を整理したものを用いて説明します。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 概要 | 車灯現示 | 歩灯現示 |
| 1 | 初期現示 | 青 | 赤 |
| 2 | 押ボタンが押された時 | 黄 | 赤 |
| 3 | 一定時間後 | 赤 | 赤 |
| 4 | 歩行者が横断可能 | 赤 | 青 |
| 5 |  | 赤 | 青点滅 |
| 6 | この後1に戻る | 赤 | 赤 |

### サイクル

サイクルとは、交通信号機の現示をひとまとめにしたものを指します。上記の表そのものがサイクルとなります。

制御機は、1つ以上のサイクルを紐づけることができます。

制御機は、1Tickごとに初回に紐づけられているサイクルを順に探索し、起動できるサイクルが発見された場合はそのサイクルを起動します。もし、現段階で起動できるサイクルが存在しなかった場合は、次回の判定に持ち越されます。

サイクルは一般的に**排他**であるべきであり、***同時間帯同条件で起動できるサイクルを2つ以上同一の制御機に登録するのは推奨されません***。この場合、どのサイクルが選ばれるかは不定です。

### フェーズ

フェーズとは、ある時点での各灯器における現示の状態を指します。上記の表で言えば、行に相当します。

サイクルは、1つ以上のフェーズを紐づけることができます。逆にいえば、最低1つはサイクルにフェーズを紐づける必要があります。空のフェーズが指定された場合、サイクルの動作が行われないどころか落ちる危険性があります。

制御機は、サイクルの起動中に1Tickごとにフェーズの終了条件をチェックします。フェーズが終了条件を満たす場合は、そのフェーズを終了し、次のフェーズに移動します。もし、現在起動しているサイクルにこれ以上続行するフェーズがない場合は、サイクルを終了します。

### チャンネル

チャンネルとは、あるフェーズにおける各灯器の現示を、種類別に分類したものを指します。上記の表で言えば、1列（車灯、歩灯）に相当します。

フェーズに対して直接現示を指定すると、1フェーズにつき1つしか現示が登録できなくなり、信号制御の意味が9割9分なくなります。そのため、チャンネルという概念を利用して、フェーズ内で異なる現示を実現することができます。

フェーズは、1つ以上のチャンネルを紐づけることができます。チャンネルを1つも登録しないということも可能ですが、そのようなフェーズはまったく意味をなさないので、特殊用途を除き通常は1つ以上を行うことになります。

上記のような概念を組み合わせることによって、現実世界で実施されている制御を完全に模倣することができます。

## 制御情報のJSONファイルについて（中級者向け）

制御機は、ArrayListによってサイクル・フェーズ・チャンネルを管理していますが、これを利用者はGUIからJSONファイルを用いてロード・セーブすることができます。拡張子が.gtsとなっていますが、中身は純粋なJSONファイルとなっています。

JSONファイルは共有用ファイルとして読み書きができるようにしているにすぎないため、1からJSONファイルを作成するのはかなり高度な知識を要求されます。内部でやっていることは、GTSCycleクラスとGTSPhaseクラスのインスタンスをもとに、そのフィールドの内容を正確に保存しているものとなります。

当社では、プリセットとしていくつかJSONファイルを作成して配布していますので、よくわからない場合はそちらを使用してください。

## カスタム制御スクリプトについて（上級者向け）

既存の組み合わせなどでは表現しきれないような非常に高度な制御を実現する場合、Javascriptを使用してサイクルをプログラミングチックに表現することができます。正確には、起動条件や次のフェーズ等を外部から直接指定することができるものになります。

なお、この機能は開発用に使用していたものを一般開放したものであり、外部のスクリプトに頼る以上当社の責任範囲を超過した者も含まれます。利用は自己責任でお願いいたします。

### カスタムスクリプトの仕様

先述の通り、Javascriptを用いて記述します。必須となる関数を定義する必要があり、この関数が存在しないとエラーでサイクルの実行がされません。また、文法エラーなどが発生した場合もサイクルの実行がされません。

|  |  |  |
| --- | --- | --- |
| 関数名 | 概要 | 戻り値 |
| canStart(te, detected, world) | このサイクルを起動できるかどうかを返す。 | サイクルが起動可能ならばtrue、不可能ならfalse |
| getNextPhase(te, detected, world) | 現在のフェーズが終了した場合に呼び出され、次のフェーズの番号を返す。フェーズの登録順に0,1,2…と割り振られており、この対応する番号を返す。番号が存在しない場合は内部で0に変換される。これ以上続行できるフェーズが存在しない場合は、-1を返す。 | 次に実行すべきフェーズのインデックス番号（int）、続行すべきものがなければ-1 |
| getInitialPhase(te, detected, world) | このサイクルが起動したとき、始めに実行されるフェーズのインデックス番号を返す。 | 初回実行時のフェーズのインデックス番号（int） |

各引数に関しては以下の通りです。

|  |  |  |  |
| --- | --- | --- | --- |
| 引数名 | 渡されるJavaの型 | 概要 | 備考 |
| te | GTSTileEntityTrafficController | このサイクルを実行している制御機のTileEntity |  |
| detected | boolean | 検知信号（押ボタン箱や感知機等）を受信しているかどうか |  |
| world | World | 制御機が設置されているワールドのインスタンス | サーバー側、クライアント側両方来る可能性があるので特定のサイドに依存する内容は避けるべき |

また、bindingという変数を使用することができます。この変数はGTSで用意したいくつかのメソッドや変数等を利用することができる、便利なオブジェクトです。

|  |  |  |  |
| --- | --- | --- | --- |
| メソッド・フィールド名 | 渡されるJavaの型 | 概要 | 備考 |
| log(any) | - | Minecraftのログに出力します。デバッグ用途が想定されています。 | 内部的にはGTSのロガーのdebugメソッドを呼び出しているにすぎません。 |
| cycle | GTSCycle | このサイクル自身です。フィールドの変更などを行うことができます。 | 多分privateメソッドは呼べません。 |

### カスタムスクリプトの例

以下は、カスタムスクリプトを用いた最小限の例です。カスタムスクリプトを使うまででもないですが、概ね以下のような形で実装します。関数外に何か処理を書くとそれも実行されますが、実行のタイミングはサイクルが登録された直後なのでご注意ください。

// サイクルは例えば夜間のみ実行可能とする  
function canStart(te, detected, world) {  
 // 現在のtickを取得  
 tick = world.getWorldTime() % 24000; // 世界の累計Tick数が返るので1日で割る  
 return tick > 18000; // 夜間に相当するTickのみ実行可能とする  
}  
  
function getNextPhase(te, detected, world) {  
 // シーケンシャルに次に進めるだけ。最後まで到達した場合は終了。  
 cycle = binding.cycle;  
 if (cycle.getNowPhaseNumber() == cycle.getPhases().size() - 1) {  
 return -1; // 到達した場合は終了  
 }  
 return cycle.getNowPhaseNumber() + 1;  
}  
  
function getInitialPhase(te, detected, world) {  
 // シーケンシャルなので初期は0  
 return 0;  
}

### カスタムスクリプトの指定方法

GTSScriptCycleをtypeに指定し、scriptPathにパスを指定すればOKです。制御機の動作中は絶対に動かさないようにしてください。可能な限り、modsディレクトリの中にあるGTSディレクトリに保存することをお勧めします。

# 交通信号機

最終更新日時：2025年6月4日

交通信号機（以下、信号機とします）は、以下のクラスにより構成されています。

* GTSBlockTrafficLight → ブロックインスタンス
* GTSTileEntityTrafficLight → 信号機のTileEntity
* GTSTileEntityRendererTrafficLight → 信号機のレンダラー
* GTSTrafficLightConfig → 信号機の個別情報（パック）

## パックJSON（コンフィグ）の仕様

信号機は、共通の項目以外に以下の設定が必要です。

|  |  |  |
| --- | --- | --- |
| 設定項目名称 | 概要 | 備考 |
| light | この信号機において、点灯するオブジェクトの一覧。文字列の配列で指定します。 | ここを設定することで、フェーズ状態において無点灯のものをしっかり描画できるようになります。 |

### textures

信号機の点灯部分、非点灯部分を別のテクスチャとして描画することにより、よりリアリティの増した描画ができるようにしています。そのため、以下の追加項目の設定が必要です。

|  |  |  |
| --- | --- | --- |
| 設定項目名称 | 概要 | 備考 |
| light | 信号機の点灯部分に関するテクスチャのパス。 | 全て同一で構わない場合は、baseと同じパスを入れても構いません。 |
| noLight | 信号機の非点灯部分に関するテクスチャのパス。 | 全て同一で構わない場合は、baseと同じパスを入れても構いません。 |

### patterns

交通信号制御機のチャンネルに指定することができるものとして指定する必要があります。ここには、この交通信号機の現示パターンを配列で指定します。配列内には、そのパターンで点灯させるモデルのオブジェクト名を指定することになります。後方互換性のため現在未使用の項目もありますが、設定をしてください。

|  |  |  |
| --- | --- | --- |
| 設定項目名称 | 概要 | 備考 |
| name | パターン名 | 半角英数字64文字以内推奨。 |
| object | このパターンで点灯させるオブジェクト名を列挙する。 | 文字列の配列で指定します。 |
| ticks | 点滅周期（Tick）。指定した周期で点滅を繰り返すようになります。なお、点灯時間・非点灯時間はそれぞれここで指定したTickの半分の時間となります。 | 0を指定すると常時点灯になります。負の数を指定しないでください。バグります。 |

### 例

{  
 "model": "models/Light\_CommonSteel.mqo",  
 "textures": {  
 "base": "NSS1\_base.png",  
 "light": "NSS1\_light.png",  
 "noLight": "NSS1\_nolight.png"  
 },  
 "body": [  
 "body",  
 "g",  
 "overg",  
 "y",  
 "overy",  
 "r",  
 "overr",  
 "body\_back",  
 "g\_back",  
 "overg\_back",  
 "y\_back",  
 "overy\_back",  
 "r\_back",  
 "overr\_back"  
 ],  
 "patterns": [  
 {  
 "name": "green",  
 "objects": [  
 "g300"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "yellow",  
 "objects": [  
 "y300"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "red",  
 "objects": [  
 "r300"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "yellow\_flush",  
 "objects": [  
 "y300"  
 ],  
 "tick": 20  
 },  
 {  
 "name": "red\_flush",  
 "objects": [  
 "r300"  
 ],  
 "tick": 20  
 },  
 {  
 "name": "green",  
 "objects": [  
 "g300",  
 "g300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "yellow",  
 "objects": [  
 "y300",  
 "y300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "red",  
 "objects": [  
 "r300",  
 "r300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "yellow\_flush",  
 "objects": [  
 "y300",  
 "y300\_back"  
 ],  
 "tick": 20  
 },  
 {  
 "name": "red\_flush",  
 "objects": [  
 "r300",  
 "r300\_back"  
 ],  
 "tick": 20  
 },  
 {  
 "name": "green\_yellow",  
 "objects": [  
 "g300",  
 "y300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "green\_red",  
 "objects": [  
 "g300",  
 "r300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "yellow\_green",  
 "objects": [  
 "y300",  
 "g300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "red\_green",  
 "objects": [  
 "r300",  
 "g300\_back"  
 ],  
 "tick": 0  
 },  
 {  
 "name": "redF\_yellowF",  
 "objects": [  
 "r300",  
 "y300\_back"  
 ],  
 "tick": 20  
 },  
 {  
 "name": "yellowF\_redF",  
 "objects": [  
 "y300",  
 "r300\_back"  
 ],  
 "tick": 20  
 }  
 ],  
 "size": 1.5,  
 "opacity": 0.5,  
 "light": [  
 "g300",  
 "y300",  
 "r300",  
 "g300\_back",  
 "y300\_back",  
 "r300\_back"  
 ],  
 "id": "NSS1\_2H33GYR\_over"

なんかかぶっているものもありますが概ねこのような形で記載します。

## チャンネルについて

交通信号制御機の欄で詳しく解説しますが、信号機はチャンネルを持つことができます。そのチャンネル名をもとに、アタッチされた制御機から信号を受信し、現示を行います。

# 交通信号柱

最終更新日時：2025年6月4日

交通信号柱（以下、ポールとします）は、以下のクラスにより構成されています。

* GTSBlockTrafficPole → ブロックインスタンス
* GTSTileEntityTrafficPole → ポールのTileEntity
* GTSTileEntityRendererTrafficPole → ポールのレンダラー
* GTSTrafficPoleConfig → ポールの個別情報（パック）

## パックJSON（コンフィグ）の仕様

ポールは、共通の項目以外に以下の設定が必要です。

|  |  |  |
| --- | --- | --- |
| 設定項目名称 | 概要 | 備考 |
| normalObject | 一般的に設置されるポールのオブジェクトを記載します。 | objectと同一でいいです |
| upJointObject | 同一のポールが上部に存在する（接続されている）場合のオブジェクトを記載します。 | 任意ですが、同じ場合でもnormalObjectと一致するものをコピペしておくと安心です。 |
| bottomJointObject | 同一のポールが下部に存在する（接続されている）場合のオブジェクトを記載します。 | 任意ですが、同じ場合でもnormalObjectと一致するものをコピペしておくと安心です。 |
| fullJointObject | 同一のポールが上下に設置されていて挟まれている場合のオブジェクトを記載します。 | 任意ですが、同じ場合でもnormalObjectと一致するものをコピペしておくと安心です。 |

なお、ポールではobject項目は使用しませんが、後方互換性のためにnormalObjectと同様のものを入れておくことを推奨します。

また、sizeは1.0想定で作成していますので、2.0などを入力すると破綻する可能性があります。少なくともジョイントは反応しません。

### 例

準備中

## モデルの作り方

ポールは、**必ず**原点を中心にして作成してください。ポールも設置時のプレイヤーの向きに影響され方向が変わるので、点対称であるべきです。

各ジョイントオブジェクトに対しても必ず全て原点で作成してください。また、サイズは全て統一してください（1つのオブジェクトだけ極端にでかい、とかを行うとそのオブジェクトに合わせて伸縮するのでカオスなことになります）。

適切に設定していれば、ポールを設置した際にモデルが切り替わり、より見た目がきれいに見えることでしょう。

# アーム

最終更新日時：2025年6月4日

アームは、以下のクラスにより構成されています。

* **GTSItemTrafficArm** → アームのアイテムインスタンス
* GTSTileEntityTrafficArm → アームのTileEntity
* GTSTrafficArmConfig → アームの個別情報（パック）

なお、他のクラスとの相違点として、以下があります。

* ブロックとして存在しません。アームは、ポールの一種となります。そのため、ブロックインスタンスではなく**アイテムインスタンス**となります。
* アームはポールと一緒に描画されるため、TileEntitySpecialRendererは存在しません。
* ブロックとして存在しない以上TileEntityの実態もないため、GTSTileEntityTrafficArmはダミーのTileEntityとなっています。すなわち、Minecraftに登録されているTileEntityではありません。

他と違いクラスが分かりにくいと思いますが、上記にしたがって閲覧いただければと思います。

## パックJSON（コンフィグ）の仕様

アームは、共通の項目以外に以下の設定が必要です。

|  |  |  |
| --- | --- | --- |
| 設定項目名称 | 概要 | 備考 |
| edgeObjects | ポール始点から描画されるアームのオブジェクトを記載します。 |  |
| baseObjects | 中間地点で描画されるアームのオブジェクトを記載します。 | 後述しますがここは必須で、割と重要なポイントです。 |
| endObjects | ゴールとなるポイントで描画されるアームのオブジェクトを記載します。 | たいていの場合はベースと同じかと思われます |
| drawStartPrimary | true / falseで指定します。trueにすると、長さが足りない場合でもedgeObjectsの描画を優先します。falseにすると、足りない場合に描画がされなくなります。 | trueを通常は指定してください。falseは特殊な用途でのみ使用します。 |

なお、アームではobject項目は使用しませんが、後方互換性のためにnormalObjectと同様のものを入れておくことを推奨します。

drawStartPrimaryに関しては、後方互換性のために入れているものとなります。通常はtrueを指定してください。コードを読んで何をやっているのか理解できる方はfalseに変更しても構いません。

### 例

{  
 "id": "Arm\_Normal2\_Ped",  
 "model": "models/Arm\_Ped2Normal.mqo",  
 "textures": {  
 "base": "arm.png"  
 },  
 "edgeObjects": ["start"],  
 "baseObjects": ["base"],  
 "endObjects": ["end"],  
 "size": 1,  
 "drawStartPrimary": false  
}

## モデルの作り方

アームはその性質上、モデルの作り方に若干の癖があります。これを守ることでよりスムーズに描画がされるので、できるだけ従ってください。コードを読んで描画内容を理解できる方は色々カスタマイズされても構いません。

### 描画の仕組み

アームの描画は、以下のようにして行われます。

1. ポールの開始地点ブロックの中心を原点とし、接続先の方向にsizeで指定した分だけの長さでedgeObjectsを描画します。
2. edgeObjectsの描画終了地点から、接続先地点のブロックからsizeで指定した分の長さを引いた長さにbaseObjectsを**スケールして**描画します。
3. 最後に、接続先地点のブロックからsizeで指定した分の長さを引いた地点から接続先地点までを、endObjectsによって描画します。

特に重要なのが2で、baseObjectsはスケーリングされて描画されます。そのため、しっかり意識して作成しないと変な伸び方をしたりします。

### baseObjectsの制約

baseObjectsは、**原点を中心としてX軸方向にスケーリングされます**。したがって、アームの作成時は必ずモデルを原点に配置し（とくにX座標）、X軸方向にアームを伸ばしてください。この時のサイズはsizeになりますのでそれも意識してください。

### その他の制約

ポールと同様に、全て原点を中心に作成してください。